Origin2000™ and Onyx2™ Performance Tuning and Optimization Guide

Document Number 007-3430-002

CONTRIBUTORS

Written by David Cortesi, based on the first edition by Jeff Fier
Illustrated by Dan Young
Edited by Christina Cary
Production by Kirsten Pekarek
Engineering contributions by David Cortesi, Leo Dagum, Wesley Jones, Eric Salo, Igor Zacharov, Marco Zagha
St. Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

© 1998, Silicon Graphics, Inc.— All Rights Reserved
The contents of this document may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND
Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

Silicon Graphics, CHALLENGE, Indy, IRIX, and Onyx are registered trademarks and the Silicon Graphics logo, InfiniteReality, O2, Onyx2, Origin200, Origin2000, POWER CHALLENGE, POWER CHALLENGE 10000, and XFS are trademarks of Silicon Graphics, Inc. CRAY is a registered trademark, and CrayLink is a trademark of Cray Research, Inc. POSIX is a trademark of IEEE. MIPS, R4000, R4400, R5000, R8000, and R10000 are registered trademarks and MIPSpro is a trademark of MIPS Technologies, Inc. NFS is a registered trademark of Sun Microsystems, Inc. X Window System is a trademark of X Consortium, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

Origin2000™ and Onyx2™ Performance Tuning and Optimization Guide
Document Number 007-3430-002

Contents

List of Examples

List of Figures

List of Tables

About This Guide

This guide tells how to tune programs that run on the Silicon Graphics Origin2000, Onyx2, and Origin200 multiprocessor systems for best performance. The material is meant for two different uses:

- As a self-paced study course, to be used by any software developer who is writing or maintaining an application to run on an Origin system.
- As an outline and supplementary material for a course delivered in person by a Silicon Graphics System Support Engineer.

The guide also contains a glossary of terms related to performance tuning and to the hardware concepts of SN0 (Scalable Node 0, the name for Silicon Graphics server architecture).

Who Can Benefit from This Guide

The guide is written for experienced programmers, familiar with IRIX commands and with either the C or Fortran programming languages. The focus is on achieving the highest possible performance by exploiting the features of IRIX, the MIPS R10000 CPU, and the SN0 architecture.

The material assumes that you know the basics of software engineering and that you are familiar with standard methods and data structures. If you are new to software design, to UNIX, to IRIX, or to Silicon Graphics hardware, this guide will not help you learn these things.

What the Guide Contains

Chapter 1, “Understanding SN0 Architecture,” describes the features of the SN0 architecture that affect performance, in particular the cache-coherent nonuniform memory architecture (CC-NUMA).

Chapter 2, “SN0 Memory Management,” reviews general programming issues for these systems and the programming practices that lead to good (and bad) performance.

Chapter 3, “Tuning for a Single Process,” covers tuning for single-process performance in detail, showing how to take best advantage of the R10000 CPU and cache memory, how to use the profiling tools, and how to select among the many compiler options.

Chapter 4, “Profiling and Analyzing Program Behavior”

Chapter 5, “Using Basic Compiler Optimizations”

Chapter 6, “Optimizing Cache Utilization”

Chapter 7, “Using Loop Nest Optimization”

Chapter 8, “Tuning for Parallel Processing,” discusses tuning issues for parallel programs, including points on how to avoid cache contention and how to distribute virtual memory segments to different nodes.

Appendix A, “Bentley’s Rules Updated,” is a summary of the performance-tuning guidelines first published by Jon Bentley in the out-of-print classic *Writing Efficient Programs*, updated for the modern world of superscalar CPUs and multiprocessors.

Appendix B, “R10000 Counter Event Types,” describes the meanings of the event counter registers in the R10000 CPU and their use for tuning.

Appendix C, “Useful Scripts and Code,” contains several longer examples and scripts mentioned in the text.

Related Documents

The material covered in this book is related to other works in the Silicon Graphics library.

Related Manuals

All of the following books can be read online on the Internet from the Tech Pubs Library at http://techpubs.sgi.com/library.

Hardware Manuals

- *MIPS R10000 Microprocessor User Guide*, Version 2.0, 007-2490-001, is the authoritative guide to the internal operations of the CPU chip used in SN0 systems.
- *Origin and Onyx2 Theory of Operations Manual*, 007-3439-*nnn*, covers the basic design of the SN0 architecture.
- *Origin and Onyx2 Programmer's Reference Manual*, 007-3410-*nnn*, has additional details of SN0 physical and virtual addressing and other topics.

Compiler Manuals

- *MIPSpro Compiling and Performance Tuning Guide*, 007-2360-*nnn*, covers compiler and linker use that is common to all the compilers, including the many optimization directives and command-line options.
- *MIPSpro 64-Bit Porting and Transition Guide*, 007-2391-*nnn*, discusses the problems that arise when porting from a 32-bit to a 64-bit computing environment, and has some discussion of optimization features.
- The Fortran compilers are documented in: *MIPSpro Fortran 77 Programmer's Guide*, 007-2361-*nnn* and *MIPSPro Fortran 90 Commands and Directives Reference Manual*, 007-3696-*nnn*. These books address general run-time issues, have some discussion of performance tuning, and document compiler directives, including the OpenMP directives for parallel processing.
- *MIPSpro C and C++ Pragmas*, 007-3587-*nnn*, covers parallelization and other directives for C programming.
- *MIPSpro Auto-Parallelizing Option Programmer's Guide*, 007-3572-*nnn*, documents how the C, C++, and Fortran compilers automatically parallelize serial programs.

Software Tool Manuals

- *SpeedShop User's Guide*, 007-3311-*nnn*, documents the tuning and profiling tools mentioned in this book.
- *Topics in IRIX Programming*, 007-2478-*nnn*, details the available models for parallel programming and documents a number of advanced programming topics.
- *Message Passing Toolkit: MPI Programmer's Manual* 007-3687-*nnn* and *Message Passing Toolkit: PVM Programmer's Manual* 007-3686-*nnn* document the use of these popular libraries for parallel programming.
- *IRIX Admin: System Configuration and Operation*, 007-2859-*nnn*, documents the commands the system administrator uses, including the system tuning variables.

Third-Party Resources

The foundation of the SN0 multiprocessor design is explained in *Scalable Shared-Memory Multiprocessing* by Daniel Lenoski and Wolf-Dietrich Weber (San Francisco: Morgan Kauffman, 1995).

A particularly good book on parallel programming is *Practical Parallel Programming* by Barr E. Bauer (Academic Press, 1992; ISBN 0120828103). Although it is not current for Silicon Graphics compilers and SN0 hardware, it has good conceptual material.

Courses and information about parallel and distributed programming are available on the Internet. The following are some useful links:

- The Boston University Scientific Computing and Visualization Group offers a number of useful tutorials on such topics as parallel programming in Fortran 90 and the use of MPI. The URL is http://scv.bu.edu/SCV/Tutorials/.
- The web page for the Computational Science and Engineering Graduate Option Program at the University of Illinois at Urbana-Champaign links to the lecture notes for courses on parallel computation and parallel numerical algorithms. The URL is http://www.cse.uiuc.edu/.
- The entire text of *Designing and Building Parallel Programs* by Ian Foster (Addison-Wesley 1995; ISBN 0-201-57594-9) is available online, with a wealth of supplementary material and links related to parallel programming. The URL is http://www.mcs.anl.gov/dbpp/.
- The NAS Parallel Benchmarks are a suite of programs that are used to compare the performance of parallel systems and parallelizing compilers. The benchmarks are described in a report found at the following URL: http://science.nas.nasa.gov/Pubs/TechReports/RNRreports/dbailey/RNR-94-007/html/npbspec.html.

Related Reference Pages

The reference pages for the compilers and tools are detailed and informative. Look up the following reference pages using the InfoSearch facility (under IRIX 6.5, found in the desktop Toolchest menu under Help > Man Pages). From the InfoSearch window you can print copies of these pages for study and for reference.

- cc(1), CC(1), f77(1), and f90(1) each document the operation and main option groups for one compiler. These pages are very similar because the most options are used by the common back-end and linker.
- ipa(5) documents the -IPA option subgroup, controlling the interprocedural analysis phase of all compilers.
- lno(5) documents the -LNO option group, controlling loop-nest optimization for all compilers.
- opt(5) documents the -OPT option group, controlling general optimizations for all compilers.
- math(3) details the standard math library used by all programs. Specially tuned libraries are described in libfastm(3) and sgimath(3) (this very long page lists all BLAS, LAPACK, and other functions).
- ld(1) documents the linker; rld(1) documents the runtime linker; and dso(5) documents the format of dynamic shared objects (runtime-linkable libraries).
- The auto-parallelizing feature is documented in auto_p(5). It generates code that uses the multiprocessing library; its runtime control is documented in mp(3F) for Fortran 77, pe_environ(5) for Fortran 90, and mp(3C) for C and C++. The newer OpenMP runtime features are in omp_threads(3), omp_lock(3), and omp_nested(3).
- The reference pages perfex(1), speedshop(1), ssrun(1), and prof(1) document the software tools used to profile and analyze your programs.

Text Conventions

Different text fonts are used in this book to indicate different kinds of information, as shown in the following table:

Terms that are defined in the Glossary (page 305).	This performance problem is generically referred to as *cache contention*.
Names of IRIX commands and command-line options.	Compile with *cc -LNO:off*. Check the CPU clock rate with *hinv*.
Names of filesystems, paths, linkable libraries, and files.	Examine */etc/config*. Devices appear in the */hw* filesystem.
Names of routines, functions and procedures when used as names.	Two common library functions are **printf()** and **wait()**.
User input and program statements or expressions, when they must be typed exactly as shown.	Use `c$doacross mp_schedtype=simple` to parallelize with the basic scheduling. Enter `y` when prompted.
Program and mathematical variables used as names; and variable elements of program expressions.	Applying *p* CPUs to a program does not result in a speedup of *p* times. The feedback file is written as *program.n.fb*.

Chapter 1

Understanding SN0 Architecture

To extract top performance from a computer system, it is important to understand the architecture of the system.This section describes the architecture of Silicon Graphics' SN0 systems: the Origin2000, Onyx2, and Origin200 systems and the components they contain, including the MIPS R10000 CPU.

Understanding Scalable Multiprocessor Memory

The architecture of SN0 (Scalable Node 0) systems the basis for a scalable multiprocessor with shared memory. The Origin2000 and Onyx2, as well as the Origin200, are based on the SN0 architecture. This section describes what a scalable, shared memory multiprocessor is, and explains the rationale for the SN0 architecture.

Memory for Multiprocessors

Because there is always a limit to the performance of a single CPU, computer manufacturers have increased the performance of their systems by incorporating multiple CPUs. Two approaches for utilizing multiple CPUs have emerged: *distributed* memory, in which each CPU has a private memory; and *shared* memory, in which all CPUs access common memory.

Previous shared memory computer designs—including the CHALLENGE and POWER CHALLENGE systems from Silicon Graphics—were implemented using a bus architecture: processor boards and memory boards plugged into a single common bus, with all communication between the hardware components occurring over the bus.

Shared Memory Multiprocessing

A shared-memory architecture has many desirable properties. Each CPU has direct and equal access to all the memory in the system. It is relatively easy to build a single operating system that uses all CPUs in a symmetrical way. Parallel programs are easy to implement on such a system. Parallelism can be achieved by inserting directives into the code to distribute the iterations of a loop among the CPUs. A Fortran loop of this type is shown in Example 1-1.

Example 1-1 Parallel Code Using Directives for Simple Scheduling

```
c$doacross local(j,k,i), shared(n,a,r,s), mp_schedtype=simple
      do j = 1, n
         do k = 1, n
            do i = 1, n
               a(i,j) = a(i,j) + r(i,k)*s(k,j)
            enddo
         enddo
      enddo
```

In Example 1-1, `mp_schedtype=simple` means that blocks of consecutive iterations are assigned to each CPU. That is, if there are *p* CPUs, the first CPU is responsible for executing the first *ceil(n/p)* iterations, the second CPU executes the next *ceil(n/p)* iterations, and so on.

One advantage of this parallelization style is that it can be done incrementally. That is, the user can identify the most time-consuming loop in the program and parallelize just that. (Loops without directives run sequentially, as usual.) The program can then be tested to validate correctness and measure performance. If a sufficient speedup has been achieved, the parallelization is complete. If not, additional loops can be parallelized one at a time.

Another advantage of the shared memory parallelization is that loops can be parallelized without being concerned about which CPUs have accessed data elements in other loops. In the parallelized loop above, the first CPU is responsible for updating the first *ceil(n/p)* columns of the array *a*.

In the loop in Example 1-2, an interleaved schedule is used, which means that the first CPU will access every *p*th column of array *a*.

Example 1-2 Parallel Code Using Directives for Interleaved Scheduling

```
c$doacross local(kb,k,t), shared(n,a,b), mp_schedtype=interleave
      do kb = 1, n
         k = n + 1 - kb
         b(k) = b(k)/a(k,k)
         t = -b(k)
         call daxpy(k-1,t,a(1,k),1,b(1),1)
      enddo
```

Because all memory is equally accessible to all CPUs, nothing special needs to be done for a CPU to access data that were last touched by another CPU; each CPU simply references the data it requires. This makes parallelization easier; the programmer needs only make sure the CPUs have an equal amount of work to do; it doesn't matter which subset of the data is assigned to a particular CPU.

Distributed Memory Multiprocessing

A different approach to multiprocessing is to build a system from many units each containing a CPU and memory. In a distributed memory architecture, there is no common bus to which memory attaches. Each CPU has its own local memory that can only be accessed directly by that CPU. For a CPU to have access to data in the local memory of another CPU, a copy of the desired data elements must be sent from one CPU to the other.

When such a distributed architecture is exposed through the software, the programmer has to specify the exchange of data. Programmed copying of data between CPUs is called message-passing. It is accomplished using a software library. MPI and PVM are two libraries that are often used to write message-passing programs.

To run a program on a message-passing machine, the programmer must decide how the data should be distributed among the local memories. Data structures that are to be operated on in parallel must be distributed. In this situation, incremental parallelism is not possible. The key loops in a program will typically reference many data structures; each structure must be distributed in order to run that loop in parallel. Once a data structure has been distributed, every section of code that references it must be run in parallel. The result is an all-or-nothing approach to parallelism on distributed memory machines. In addition, often a data structure is bigger than a single local memory, and it may not be possible to run the program on the machine without first parallelizing it.

In the preceding section you saw that different loops of a program may be parallelized in different ways to ensure an even allocation of work among the CPUs. To make a comparable change on a distributed memory machine, data structures have to be distributed differently in different sections of the program. Redistributing the data structures is the responsibility of the programmer, who must find efficient ways to accomplish the needed data shuffling. Such issues never arise in the shared memory implementation.

The bottom line is that programming, and especially parallel programming, is much easier in a shared memory architecture than in a distributed memory architecture.

Scalability in Multiprocessors

Why would anyone build anything but a shared memory computer? The reason is *scalability*. Table 1-1 shows the performance of the NAS parallel benchmark kernel running on a POWER CHALLENGE 10000 system. Results are presented for a range of CPU counts and for two problem sizes: Class A represents a small problem size, and the Class B problem is four times larger. The columns headed "Amdahl" show the theoretical speedup predicted by Amdahl's law (see "Understanding Parallel Speedup and Amdahl's Law" on page 188).

Table 1-1 Scaling of NAS Parallel Benchmark on a Bus Architecture

#CPUs	Class A	Speedup	Amdahl	Class B	Speedup	Amdahl
1	62.73	1.00	1.00	877.74	1.00	1.00
2	32.25	1.95	1.95	442.66	1.98	1.98
4	16.79	3.74	3.69	225.10	3.90	3.90
8	9.58	6.55	6.68	125.71	6.98	7.54
12	7.96	7.88	9.16	103.44	8.49	10.96
16	7.55	8.31	11.24	98.92	8.87	14.16

Table 1-1 shows that the performance scales well with number of CPUs, through eight CPUs. Then it levels off; the 12-CPU performance is well short of 150% of the 8-CPU performance, and the 16-CPU times are only slightly improved. The cause is the common bus in the shared memory system. It has a finite bandwidth and hence a finite number of transactions that it can transport per second. For this particular program, the limit is reached when 12 CPUs are accessing memory simultaneously. The bus is said to be "saturated" with memory accesses; and when 16 CPUs run the program, the execution time remains almost the same. For other programs, the number of CPUs needed to saturate the bus varies. But once this limit has been reached, adding more CPUs will not significantly increase the program's performance—the system lacks *scalability* past that limit.

Scalability and Shared, Distributed Memory

Although a shared memory architecture offers programming simplicity, the finite bandwidth of a common bus can limit scalability. A distributed memory, message-passing architecture cures the scalability problem by eliminating the bus, but it also eliminates the programming simplicity of the shared memory model. The SN0 scalable shared memory architecture aims to deliver the best of both approaches.

In the SN0 design, memory is physically distributed; there is no common bus that could become a bottleneck. Memory is added with each CPU and memory bandwidth grows as CPUs and memory are added to the system. Nevertheless, SN0 hardware presents all memory to software within a unified, global address space. As far as software is concerned, all memory is shared, just as in a bus-based system. The programming ease is preserved, but the performance is scalable.

The improvement in scalability is easily observed in practice. Table 1-2 shows the performance of the NAS parallel benchmark (FT kernel) on an Origin2000 system.

Table 1-2 Scaling of NAS Parallel Benchmark on the Origin2000

#CPUs	Class A	SpeedUp	Class B	Speedup
1	55.43	1.00	810.41	1.00
2	29.53	1.88	419.72	1.93
4	15.56	3.56	215.95	3.75
8	8.05	6.89	108.42	7.47
16	4.55	12.18	54.88	14.77
32	3.16	17.54	28.66	28.28

You can see that programs with sufficient parallelism now scale to as many CPUs as are available. What's more, this is achieved with the same code that was used on the bus-based shared memory system.

In addition, the per-CPU performance on Origin2000 systems is better than on POWER CHALLENGE 10000 systems (compare the first four rows of Table 1-2 with Table 1-1) because of the improved memory access times of the SN0 system.

Understanding Scalable Shared Memory

To understand scalability in the SN0 architecture, first look at how the building blocks of an SN0 system are connected.

SN0 Organization

Figure 1-1 shows high-level block diagrams of SN0 systems with various numbers of CPUs. Each “N” square represents a *node* board, containing CPUs and some amount of memory. Each “R” circle represents a *router*, a board that routes data between nodes.

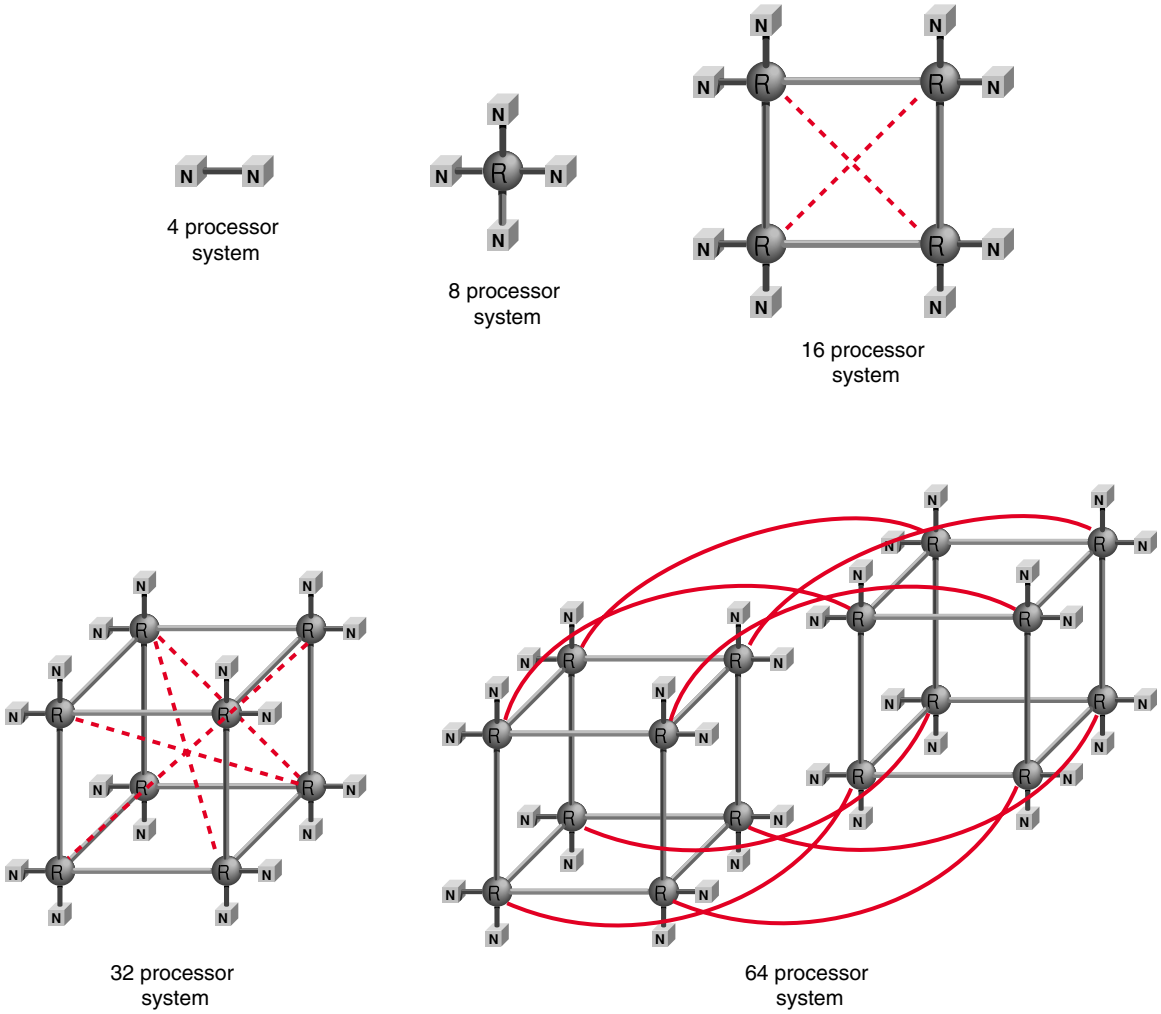


Figure 1-1 Block Diagrams of 4-, 8-, 16-, 32-, and 64-CPU SN0 Systems

Each SN0 node contains one or two CPUs, some memory, and a custom circuit called the *hub*. Additionally, I/O can be connected to a node through one or more XIO boards. The hub in a node directs the flow of data between CPUs, memory, and I/O. Through the hub, memory in a node can be used concurrently by one or both CPUs, by I/O attached to the node, and—via a router—by the hubs in other nodes.

The smallest SN0 systems consist of a single node. For example, an Origin200 system is a deskside unit containing a single node, with one or two CPUs, memory, and I/O.

Larger SN0 systems are built by connecting multiple nodes. The first diagram in Figure 1-1 shows a two-node system, such as an Onyx2 Deskside system. Connecting two nodes means connecting their hub chips. In a two-node system this requires only wiring the two hubs together. The hub on either node can access its local memory, and can also access memory in the other node, through the other hub.

A hub determines whether a memory request is local or remote based on the physical address of the data. (The node number is encoded in the high-order bits of the 64-bit physical address.) Access to memory on a different node takes more time than access to memory on the local node, because the request must be processed through two hubs.

When there are more than two nodes in a system, their hubs cannot simply be wired together. The hardware in an SN0 system that connects nodes is called a *router*. A router can be connected to up to six hubs or other routers. In the 8-CPU system shown in Figure 1-1, a single router connects four hubs. The router, like a hub, introduces a tiny delay in data transfer, so a CPU in one node incurs an extra delay in accessing memory on a different node.

Now compare the 16-, 32- and 64-CPU system diagrams. From these diagrams you can begin to see how the router configurations scale: each router is connected to two hubs, routers are then connected to each other forming a binary *n*-cube, or *hypercube*, in which *n*, the dimensionality of the router configuration, is the base-2 logarithm of the number of routers.

SN0 Memory Distribution

The key point is that the SN0 hardware allows the memory to be physically distributed and yet shared by all software, just as in a bus-based system. Each node contains some local memory (usually many megabytes) that is accessed in the least time. The hardware makes all memory equally accessible from a software standpoint, by routing memory requests through routers to the other nodes. However, the hypercube topology ensures that these data paths do not grow linearly with the number of CPUs: as more nodes and routers are added, more parallel data paths are created. Thus a system can be scaled up without fear that the router connections will become a bottleneck, the way the bus becomes a bottleneck in a bus architecture.

One nice characteristic of the bus-based shared memory systems has been sacrificed: the access time to memory is no longer uniform, but varies depending on how far away the memory lies from the CPU. The two CPUs in each node have quick access through their hub to their local memory. Accessing remote memory through an additional hub adds an extra increment of time, as does each router the data must travel through. This is called nonuniform memory access (NUMA). However, several factors combine to smooth out access times:

1. The hardware has been designed so that the incremental costs to access remote memory are not large. The choice of a hypercube router configuration means that the number of routers information must pass through is at most n+1, where n is the dimension of the hypercube; this grows only as the logarithm of the number of routers.

   As a result, the *average* memory access time on even the largest SN0 system is no greater than the *uniform* memory access time on a POWER CHALLENGE 10000 system.

2. The R10000 CPUs operate on data that are resident in their caches. When programs use the caches effectively, the access time to memory (local or remote) is unimportant because the great majority of accesses are satisfied from the caches.

3. The R10000 CPUs can prefetch data that are not in cache. Other work can be carried out while these data move from local or remote memory into the cache, thus hiding the access time.

4. Through operating system support or explicit programming, the data of most programs can be made to reside primarily in memory that is local to the CPUs that execute the code.

The SN0 architecture provides shared memory hardware without the limitations of traditional bus-based designs. The following topics look in detail at the components that make up the system.

SN0 Node Board

An Origin2000 or Onyx2 system starts with a node board. Figure 1-2 displays a block diagram of a node board and the actual appearance of one.

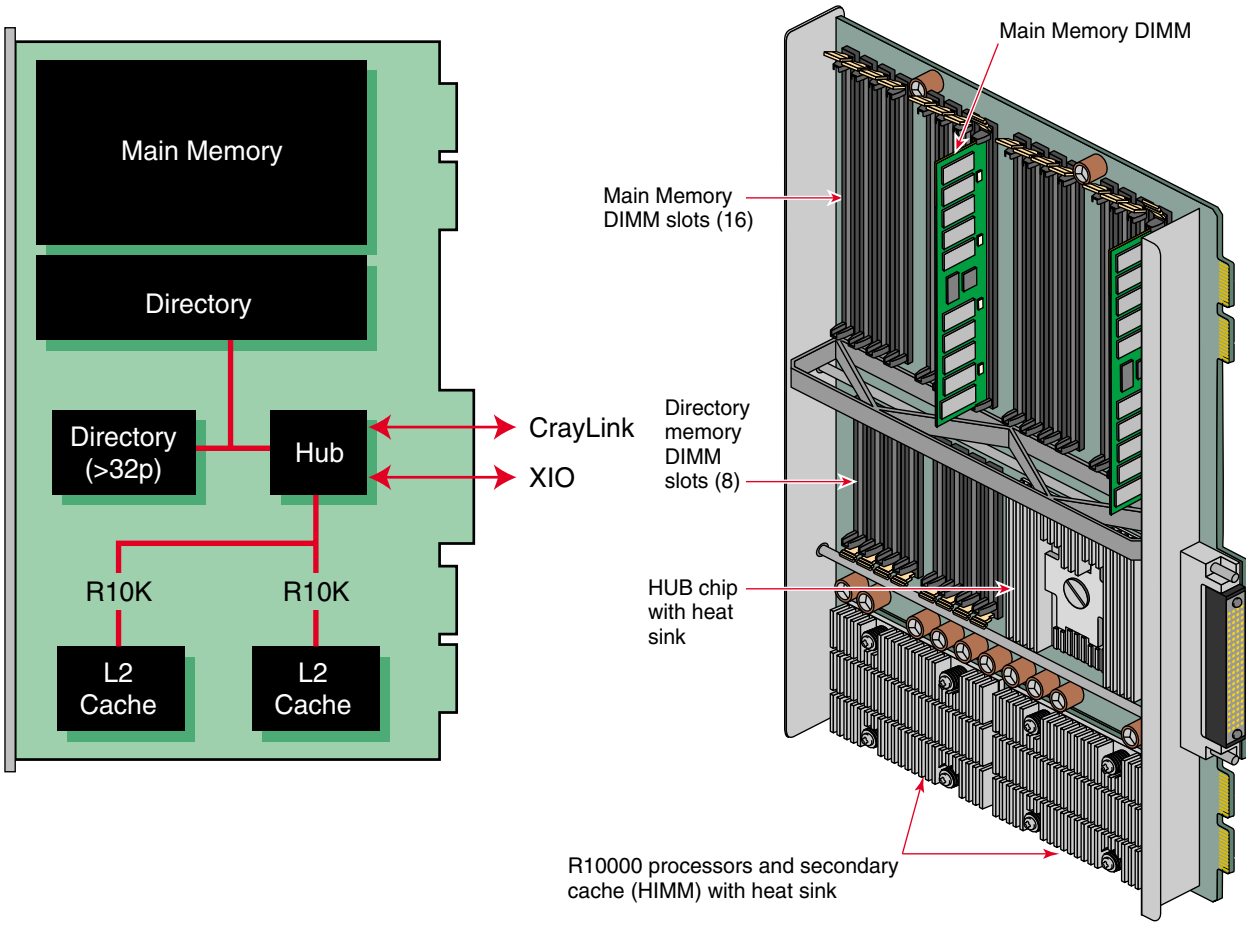


Figure 1-2 Block Diagram and Approximate Appearance of a Node Board

CPUs and Memory

Each node contains two R10000 CPUs. (An overview of the R10000 CPU appears later; see "Understanding MIPS R10000 Architecture" on page 20.) Node boards are available with CPUs that run at different clock rates, from 180 MHz up to 250 MHz, and using different sizes of secondary cache. (The *hinv* command lists these details for a given system; see the hinv(1) reference page.) All the nodes in a system must run at the same clock rate.

In addition to CPUs, a node board has sockets to hold memory DIMMs. A node board can be configured with 64 MB to 4 GB of memory.

Memory Overhead Bits

In addition to data memory, each main memory DIMM contains extra memory bits. Some of these are used to provide single-bit error correction and dual-bit error detection. The remaining bits store what is known as the *cache directory*. (The operation of this directory-based cache is described in more detail in a following section; see "Understanding Cache Coherency" on page 12.) The use of the cache directory means that the main memory DIMMs must store additional information proportional to the number of nodes in the system. In systems with 32 or fewer CPUs, the directory overhead amounts to less than 6 percent of the storage on a DIMM; for larger systems, it is less than 15 percent. This is comparable to the overhead required for the error correction bits.

SN0 systems can scale to a large number of nodes, and a significant amount of directory information could be required to accommodate the largest configuration. In order not to burden smaller systems with unneeded directory memory, the main memory DIMMs contain only enough directory storage for systems with up to 16 nodes (i.e., 32 CPUs). For larger systems, additional directory memory is installed in separate directory DIMMs. Sockets for these DIMMS are provided below the main memory DIMMs, as can be seen in Figure 1-2.

Hub and CrayLink

The final component of the node board is the hub. As described in "Understanding Scalable Shared Memory" on page 6, the hub controls data traffic between the CPUs, memory and I/O. The hub has a direct connection to the main memory on its node board. This connection provides a raw memory bandwidth of 780 MBps to be shared by the two CPUs on the node board. In practice, data cannot be transferred on every bus cycle, so the achievable bandwidth is more like 600 MBps.

Access to memory on other nodes is through a separate connection called the CrayLink interconnect, which attaches to either a router or another hub. The CrayLink network is bidirectional, and has a raw bandwidth of 780 MBps in each direction. The effective bandwidth achieved is about 600 MBps in each direction, because not all information sent between nodes is user data. To request a copy of a cache line from a remote memory, a hub must first send a 16-byte address packet to the hub that manages the remote memory; this specifies which cache line is desired and where it needs to be sent. Then, when the data are returned, along with the 128-byte cache line of user data, another 16-byte address header is sent (so the receiving hub can distinguish this cache line from others it may have requested). Thus, 16 + 128 + 16 = 160 bytes of data are passed through the CrayLink interconnect to transfer the 128-byte cache line—so the effective bandwidth is 780 × (128 ÷ 160) = 624 MBps in each direction.

XIO Connection

The hub also controls access to I/O devices through a channel called XIO. This connection has the same bandwidth as the CrayLink connection, but different protocols are used in communicating data. The connection to multiple I/O devices using XIO is described later, under "SN0 Input/Output" on page 16.

Understanding Cache Coherency

Each CPU in an SN0 system has a secondary cache memory of 1 MB or 4 MB. The CPU only fetches and stores data in its cache. When the CPU must refer to memory that is not present in the cache, there is a delay while a copy of the data is fetched from memory into the cache. More details on CPU use of cache memory, and its importance to good performance, is covered in "Cache Architecture" on page 22.

The point here is that there could be as many independent copies of one memory location as there are CPUs in the system. If every CPU refers to the same memory address, every CPU's cache will receive a copy of that address. (This can occur when running a parallelized loop that refers to a global variable, for example. It is often the case with some kernel data structures.)

So long as all CPUs only examine the data, all is well; each can use its cached copy. But what if one CPU then modifies that data by storing a new value in it? All the other cached copies of the location instantly become invalid. The other CPUs must be prevented from using what is now "stale data." This is the issue of cache coherence: how to ensure that all caches reflect only the true state of memory.

Coherency Methods

Cache coherence is not the responsibility of software (except for kernel device drivers, which must take explicit steps to keep I/O buffers coherent). For performance to be acceptable, cache coherence must be managed in hardware. Furthermore, the cache coherence hardware is external to the R10000 CPU. (The CPU specifies only what the cache must do, not how it does it. The R10000 is used with several different kinds of secondary cache in different systems.)

In the SN0 architecture, cache coherence is the responsibility of the hub chip.

The cache coherence solution in SN0 is fundamentally different from that used in the earlier systems. Because they use a central bus, CHALLENGE and Onyx systems can use a *snoopy cache*—one in which each CPU board observes every memory access that moves on the bus. When a CPU observes another CPU modifying memory, the first CPU can automatically invalidate and discard its now-stale copy of the changed data.

The SN0 architecture has no central bus, and there is no efficient way for a CPU in one node to know when a CPU in a different node modifies memory on its node or memory in yet a third node. Any scheme that would broadcast memory accesses to every node would cause coherency information to grow as the square of the number of nodes, defeating scalability.

Instead, SN0 uses a different scheme based on a *cache directory*, which permits cache coherence overhead to scale slowly. (For a theoretical discussion of directory-based caches, see the book by Lenoski and Weber listed under "Related Documents" on page xxix.)

Understanding Directory-Based Coherency

Briefly, here's how directory-based coherency works. As suggested by the diagram in Figure 1-3, the memory in any node is organized as an array of *cache line*s of 128 bytes. Along with the data bits for each cache line is an extra set of bits, one bit per node. (In large systems, those with more than 128 CPUs, a directory bit represents more than one node, but the principle is the same.) In addition, each directory line contains one small integer, the number of a node that owns that cache line of data exclusively.

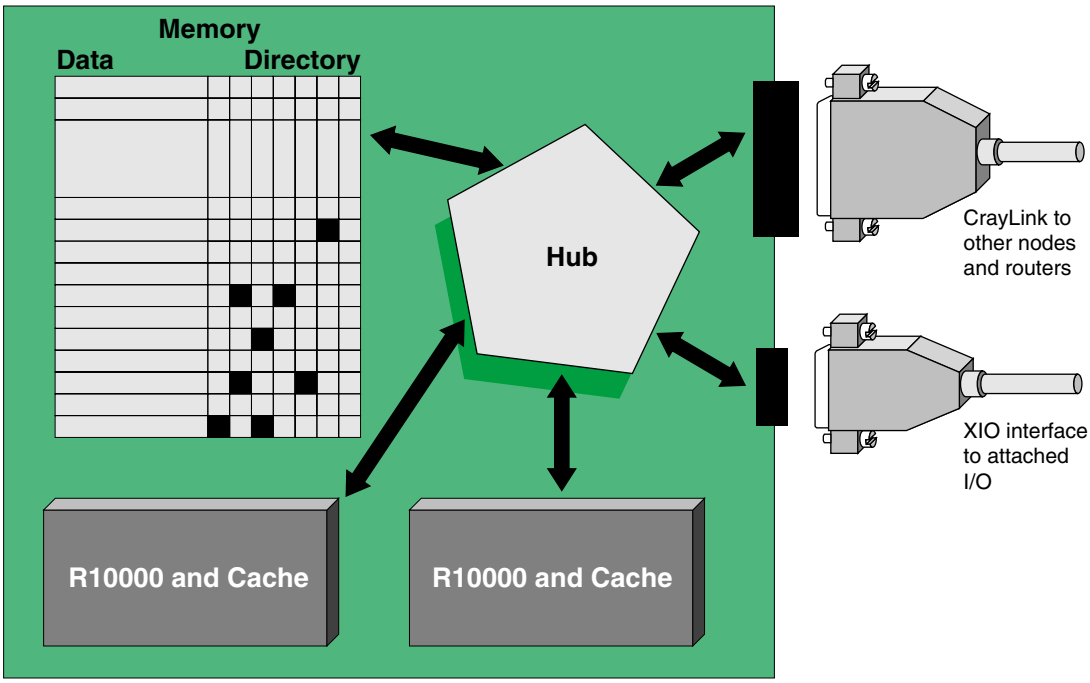


Figure 1-3 Block Diagram of Memory, Hub, and Cache Directory

Whenever the hub receives a request for a cache line of data— the request can come from a CPU on the same node, from an attached I/O device, or from some other node via the Craylink—the hub checks that line's directory to make sure it is not owned exclusively. Normally it is not, and the hub fetches the whole, 128-byte line and sends it to the requestor.

At the same time, the hub sets on the directory bit that corresponds to the requesting node. The requesting node is this current node, if the request came from a CPU or I/O device. When the request comes over the Craylink, the requesting node is some other node in the system.

For any line of memory, as many bits can be set as there are nodes in the system. Directory bits are set in parallel with the act of fetching memory; there is no time penalty for maintaining the directory. As long as all nodes that use a cache line only read it, all nodes operate in parallel with no delays.

Modifying Shared Data

When a CPU wants to modify a cache line, it must first gain exclusive ownership. The CPU sends an ownership request to the hub that manages that memory. Whenever the hub receives an ownership request—whether the request comes from a CPU on the same node, or from some other node via the Craylink—the hub examines the directory bits of that cache line. It sends an invalidation message to each node for which a bit is set—that is, to each node that has made a copy of the line. Only when this has been done does the hub set the number of the updating node in the directory for that line as the exclusive owner. Then it responds to the requesting node, and that CPU can complete its store operation.

When a hub receives an invalidation message, it removes that cache line from the cache of either (or both) CPUs on its node. In the event those CPUs attempt to refer to that cache line again, they will have to request a new copy, rather than working on the old data.

Typically, invalidations need to be sent to only a small number of other nodes; thus the coherency traffic only grows proportionally to the number of nodes, and there is sufficient bandwidth to allow the system to scale.

When a CPU no longer needs a cache line (for example, when it wants to reuse the cache space for other data), it notifies its hub. The hub sends a release message to the hub that owns the line (if the line is in a different node). If it has exclusive ownership, the hub also sends an updated copy of the cache line. The hub that manages the line removes the exclusive access (if it was set), stores the modified data (if it was modified), and clears the directory bit for the releasing node.

Reading Modified Data

When a CPU wants to read a cache line that is exclusively owned, a more complicated sequence occurs. The request for the line comes to the hub that manages that memory, as usual. That hub observes that some other node has exclusive ownership of the line. Presumably, that owner has modified the data in its cache, so the true, current value of the data is there, not in memory.

The managing hub requests a copy of the line from the owning node. That (owner) hub sends the data to the requesting hub. This retrieves the latest data without waiting for it to be written back to memory.

Other Protocols

The SN0 cache coherency design requires other protocols. There is a protocol by which CPUs can exchange exclusive control of a line when both are trying to update it. There are protocols that allow kernel software to invalidate all cached copies of a range of addresses in every node. (The IRIX kernel uses this to invalidate entire pages of virtual memory, so that it can transfer ownership of virtual pages from one node to another.)

Memory Contention

The directory-based cache coherency mechanism uses a lot of dedicated circuitry in the hub to ensure that many CPUs can use the same memory, without race conditions, at high bandwidth. As long as memory reads far exceed memory writes (the normal situation), there is no performance cost for maintaining coherence.

However, when two or more CPUs alternately and repeatedly update the same cache line, performance suffers, because every time either CPU refers to that memory, a copy of the cache line must be obtained from the other CPU. This performance problem is generically referred to as *cache contention*. Two variations of it are:

- *memory contention*, in which two (or more) CPUs try to update the same variables
- *false sharing*, in which the CPUs update distinct variables that only coincidentally occupy the same cache line

Memory contention occurs because of the design of the algorithm; correcting it usually involves an algorithmic change. False sharing is contention that arises by coincidence, not design; it can usually be corrected by modifying data structures.

Part of performance tuning of parallel programs is recognizing cache coherency contention and eliminating it.

SN0 Input/Output

As mentioned, the hub controls access to I/O devices as well as memory. Each hub has one XIO port. To allow connections to multiple I/O devices, each hub is connected to a Crossbow (XBOW) I/O controller chip. Up to two hubs may be connected to the same XBOW.

The XBOW is a dynamic crossbar switch that allows up to two hubs to connect to six I/O bus attachments in any combination, as shown in Figure 1-4.

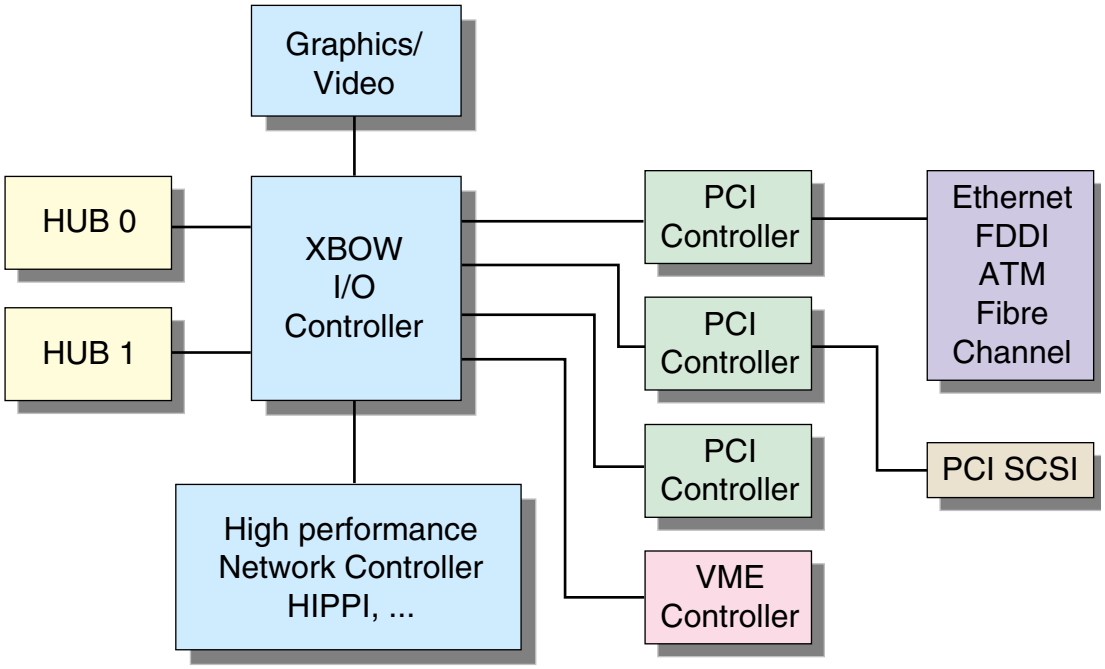


Figure 1-4 XIO and XBOW Provide I/O Attachment to a Node

I/O Connections and Bandwidth

Each XIO connection, from hub to XBOW and from XBOW to I/O device, is bidirectional with a raw bandwidth of 780 MBps in each direction. Like the CrayLink interconnect, some of the bandwidth is used for control information, so achievable bandwidth is approximately 600 MBps in each direction.

XIO connections to I/O devices are 16 bits wide, although an 8-bit subset may be used for devices for which half the XIO bandwidth is sufficient. Currently, only Silicon Graphics graphics devices utilize the full 16-bit connection. Industry-standard I/O interfaces such as Ethernet, FDDI, ATM, Fibre Channel, HiPPI, SCSI, and PCI operate at more modest bandwidths, so XIO cards supporting these devices employ the 8-bit subset. Each standard interface card first converts the 8-bit XIO subset to the PCI bus using a custom chip called the PCI adapter. Standard hardware is then employed to convert the PCI bus to the desired interface: VME, Fibrechannel, and so on.

One particular interface card, the IO6, must be installed in every Origin2000 and Onyx2 system. This I/O card supplies two SCSI buses to provide the minimal I/O requirements of a system; namely, it allows the connection of a CD-ROM drive (for loading system software) and a system disk. It also contains serial and Ethernet ports to allow attachment of a console and external network.

I/O Access to Memory

When two nodes are connected to a XBOW, control of the 6 I/O devices is statically partitioned between the two nodes at system boot time. If one node is inoperable, the second can be programmed to take control of all of the devices.

From the standpoint of the XBOW chip, the hub is a portal to and from memory. From the standpoint of the hub, the XBOW is another device that accesses memory, just like a CPU. Software programs the I/O devices to send data from memory out to the device or to read data from the device into memory. In both cases, the device, through the XBOW, requests access to cache lines of memory data.

These requests go to the hub to which the XBOW is attached. If the memory requested is on that node, it is returned. If it is not, the hub passes the request over the Craylink interconnect, to get the memory from the node where it resides—exactly as for a CPU's request for memory. In this way, an I/O device attached to any node can read and write to memory anywhere in the system.

SN0 Latencies and Bandwidths

We conclude the discussion of the SN0 architecture by summarizing the bandwidths and latencies of different-sized systems. These are shown in Table 1-3.

Table 1-3 Bisection Bandwidth and Memory Latency

Configuration	Bisection Bandwidth		Router Hops		Read Latency	
Nodes (CPUs)	**System**	**Per CPU**	**Maximum**	**Average**	**Max**	**Average**
1 (2)	n.a.	n.a.	0	0	313	313
2 (4)	620	310	0	0	497	405
4 (8)	620	310	1	.75	601	528
8 (16)	620	310	2	1.63	703	641
16 (32)	620	310	3	2.19	805	710
32 (64)	310	160	5	2.97	1010	796
64 (128)	310	160	6	3.98	1112	903

The meanings of the columns of Table 1-3 are as follows:

1. The number of nodes and CPUs in the system. Larger SN0 systems can be configured in a variety of ways; this table assumes that the optimum arrangement of nodes and routers has been used.
2. The bisection bandwidth is the total amount of data that can flow in the system concurrently, in megabytes per second of user data. The term “bisection” comes from imagining that the system is bisected by a plane, and measuring the data that crosses that plane.

   Bisection bandwidth per CPU is the portion of the total bandwidth that one CPU could use in principle.
3. Router hops is the number of routers that could handle a request for memory data. The maximum column shows the longest path; the average column is the average over all node-to-node paths. For example, in an 8-CPU system (with a star router), each node can access its own memory with zero router hops, but it takes one router hop to get to any of the three remote memories. The average number of hops is thus 0.75.

4. Read latency is the time to access the first word of a cache line read from memory, in nanoseconds. For local memory, the latency is 313 nanoseconds. For a hub-to-hub direct connection (that is, a two-node configuration), the maximum latency is 497 nsec. For larger configurations, the maximum latency grows approximately 100 nsec for each router hop.

   Average latency is calculated by averaging the latencies to all local and remote memories. What is amazing is that even for the largest configuration, 128 CPUs, the average latency is no worse than that of a POWER CHALLENGE system, demonstrating that shared memory can be made scalable without negatively affecting performance.

Understanding MIPS R10000 Architecture

This section describes the features of the MIPS R10000 CPU that are important for performance tuning.

Superscalar CPU Features

The MIPS R10000, designed to solve many of the performance bottlenecks common to earlier microprocessors, is the CPU used in SN0 systems. SN0 systems have shipped with CPUs running at various clock rates from 195 MHz to 235 MHz.

The R10000 is a four-way *superscalar* RISC CPU. “Four-way” means that it can fetch and decode four instructions per clock cycle. “Superscalar” means that it has enough independent, pipelined execution units that it can complete more than one instruction per clock cycle. The R10000 contains:

- A nonblocking load-store unit that manages memory access.
- Two 64-bit integer Arithmetic/Logic Units (ALUs) for address computation and for arithmetic and logical operations on integers.
- A pipelined floating point adder for 32- and 64-bit operands.
- A pipelined floating point multiplier for 32- and 64-bit operands.

The two integer ALUs are not identical. Although both perform add, subtract, and logical operations, one can handle shifts and conditional branch and conditional move instructions, while the other can execute integer multiplies and divides. Similarly, instructions are partitioned between the floating point units. The floating-point adder is responsible for add, subtract, absolute value, negate, round, truncate, ceiling, floor, conversions, and compare operations. The floating-point multiplier carries out multiplication, division, reciprocal, square root, reciprocal square root, and conditional move instructions.

The two floating-point units can be chained together to perform multiply-then-add and multiply-then-subtract operations, which are single instruction codes. These combined operations, often referred to by their operation codes of madd and msub, are designed to speed the execution of code that evaluates polynomials.

The R10000 can complete most single instructions in one or two clock cycles. However, integer multiple and divide, and floating-point divide and square-root instructions, can occupy one of the ALUs for 20 to 35 clock cycles. Because a number of instructions are in the CPU, being decoded and executed in parallel, the R10000 routinely averages more than one completed instruction per clock cycle, and can reache as many as two instructions per clock cycle in some highly tuned loops.

MIPS IV Instruction Set Architecture

The R10000 implements the MIPS IV instruction set architecture (ISA). This is the same ISA supported by the MIPS R8000, the CPU used in the Silicon Graphics POWER CHALLENGE series, so programs compiled for the older systems are binary-compatible with newer ones. The MIPS IV ISA is a superset of the previous MIPS I, II, and III ISAs that were used in several preceding generations of Silicon Graphics workstations and servers, so programs compiled for those systems are also binary-compatible with the R10000.

The MIPS IV ISA augmented the previous instruction sets with:

- Floating point multiply-add (madd and msub) instructions, and reciprocal and reciprocal square root instructions
- Indexed loads and stores, which allow coding more efficient loops
- Prefetch instructions, which allow the program to request fetching of a cache line in advance of its use, so memory-fetch can be overlapped with execution
- Conditional move instructions, which can replace branches inside loops, thus allowing superscalar code to be generated for those loops

This book assumes your programs are written in a high-level language, so you will not be coding machine language yourself. However, the C and Fortran compilers, when told to use the MIPS IV ISA, do take advantage of all these instructions to generate faster code. For this reason it can be worthwhile to recompile an older program.

For more details on the MIPS IV ISA, see the mips4(5) reference page, and the *MIPS R10000 Microprocessor User Guide* listed under "Related Manuals" on page xxix.

Cache Architecture

The R10000 uses a two-level cache hierarchy: a level-1 (L1) cache internal to the CPU chip, and a level-2 (L2) cache external to it. Both L1 and L2 data caches are nonblocking. That is, the CPU does not stall on a cache miss—it suspends the instruction and works on other instructions. As many as four outstanding cache misses from the combined two levels of cache are supported.

Level-1 Cache

Located on the CPU chip are:

- A 32 KB, two-way set associative, instruction cache, managed as an array of 512, 64-byte cache lines.
- A 32 KB, two-way set associative, two-way interleaved data cache, managed as an array of 1024, 32-byte cache lines.

These two on-chip caches are collectively called the L1, cache. Whenever the CPU needs an instruction, it looks first in the L1 instruction cache; whenever it needs a data operand it looks first in the L1 data cache. When the data is found there, no memory request is made external to the CPU. The ultimate in speed is achieved when the entire working set of a loop fits in the 64 KB L1 cache. Unfortunately, few time-consuming programs fit in so little memory.

The R10000 CPU can keep exact counts of L1 cache hits and misses. You can take a profile of your program to measure its L1 cache behavior.

Level-Two Cache

When the CPU fails to find memory in the L1 cache, it looks next in an off-chip cache consisting of chips on the node board. The L2 cache is a two-way set associative, unified (instructions and data) cache, usually 4 MB in current SN0 systems. The R10000 CPU permits the line size of the L2 cache to be either 64 B or 128 B, but all SN0 systems use 128-byte cache lines. Both the L1 data cache and the L2 unified cache employ a least recently used (LRU) replacement policy for selecting in which set of the cache to place a new cache line.

The L2 cache can be run at various clock rates, ranging from the same as the CPU down to one-third of that frequency. In current SN0 systems it operates at two-thirds of the CPU frequency.

The R10000 CPU is rare among CPU designs in supporting a set associative off-chip cache. To provide a cost-effective implementation, however, only enough chip terminals are provided to check for a cache hit in one set of the secondary cache at a time. To allow for two-way functionality, an 8,192-entry way prediction table is used to record which set of a particular cache address was most recently used. This set is checked first to determine whether there is a cache hit. This takes one cycle and is performed concurrently with the transfer of the first half of the set's cache line. The other set is checked on the next cycle while the second half of the cache line for the first set is being transferred. If there is a hit in the first set, no extra cache access time is incurred. If the hit occurs in the second set, its data must be read from the cache and a minimum four-cycle mispredict penalty is incurred. The net effect is that the time to access the L2 cache is variable, although always much faster than main-memory access.

Cache miss latency to the second-level cache—that is, the time that an instruction is blocked waiting for data from the L2 cache—is from 8 to 10 CPU clock cycles, when the data is found in the L2 cache and the way is correctly predicted.

The R10000 CPU can keep exact counts of L2 cache hits and misses. You can take a profile of your program to measure its L2 cache behavior.

Out-of-Order and Speculative Execution

The chief constraint on CPU speed is that the speed of memory is so much slower: a superscalar CPU running at 200 MHz can complete more than one instruction in five nanoseconds. Compare this to the memory latencies shown in Table 1-3 on page 19. Potentially the CPU could spend the majority of its time waiting for memory fetches. The R10000 CPU uses out-of-order execution and speculative execution in conjunction with its nonblocking caches to try to hide this latency.

Executing Out of Order

From the programmer's point of view, instructions must be executed in predictable, sequential order; when the output of an instruction is written into its destination register, it is immediately available for use in subsequent instructions. Pipelined superscalar CPUs, however, execute several instructions concurrently. The result of a particular instruction may not be available for several cycles. Often, the next instruction in program sequence must wait for its operands to become available, while instructions that come later in the sequence are ready to be executed.

The R10000 CPU dynamically executes instructions as their operands become available—out of order if necessary. This out-of-order execution is invisible to the programmer. What might make out-of-order execution dangerously visible is the possibility of an exception, such as a page fault or a divide-by-zero. Suppose some instructions have been completed out of order when another instruction, which nominally preceded them in the program code, causes an exception. The program state would be ambiguous.

The R10000 CPU avoids this danger by making sure that any result that is generated out of order is temporary until all previous instructions have been successfully completed. If an exception is generated by a preceding instruction, the temporary results can be undone and the CPU state returned to what it would have been, had the instructions been executed sequentially. When no exceptions occur, an instruction executed out of order is "graduated" once all previous instructions have completed; then its result is added to the visible state of the CPU.

Queued and Active Instructions

Up to 32 instructions may be active in the CPU at a given time, enough to hide at least eight cycles of latency. These instructions are tracked in an active list. Instructions are added to the active list when they are decoded and are removed upon graduation.

Queues are used to select which instructions to issue dynamically to the execution units. Three separate 16-entry queues are maintained: an integer queue for ALU instructions, a floating-point queue for FPU instructions, and an address queue for load and store instructions. Instructions are placed on the queues when they are decoded. Integer and floating-point instructions are removed from their respective queues once the instructions have been issued to the appropriate execution unit. Load and store instructions remain in the address queue until they graduate, because these instructions can fail and may need to be reissued; for example, a data cache miss requires the instruction to be reissued.

Speculative Execution

When the CPU encounters a conditional branch, the input operand of the branch may not be ready, in which case the CPU does not know which instruction should follow the branch. Older pipelined CPUs had no choice but to stall when they encountered a branch, and wait until the branch operand was available before continuing.

The R10000 CPU instead makes a guess: it predicts which way the branch will likely go, and continues executing instructions speculatively along that path. That is, it predicts whether or not the branch will be taken, and fetches and executes subsequent instructions accordingly. If the prediction is correct, execution proceeds without interruption. If the prediction is wrong, however, any instructions executed down the incorrect program path must be canceled, and the CPU must be returned to the state it was in when it encountered the mispredicted branch.

The R10000 may speculate on the direction of branches nested four deep. To predict whether or not to take the branch, a 2-bit algorithm, based on a 512-entry branch history table, is used. Simulations have shown the algorithm to be correct 87% of the time on the SpecInt92 suite.

The R10000 CPU can keep exact counts of branch prediction hits and misses. You can take a profile of your program to measure how many branches it executed and what percentage were mispredicted.

Summary

The SN0 architecture and the MIPS R10000 CPU together make up a very sophisticated computing environment:

- At the micro level of a single instruction stream, the CPU queues as many as 32 instructions, executing them out of order as their operands become available from memory, and speculatively predicting the direction that branches will take. It also manages two levels of cache memory.
- At the memory level, the system caches memory at multiple levels: the on-chip L1 cache; the off-chip L2 cache of 4MB; the local node's main memory of 64 MB or more (usually much more); and the total memory of every node in the system—up to 63 other nodes, all equally accessible in varying amounts of time.
- At the node level, the hub chip directs the flow of multiple, 780 MBpsec, streams of data passing between memory, two L2 caches, an XBOW chip, and a Craylink connection that may carry requests to and from 63 other hubs.
- At the system level, multiple I/O buses of different protocols—PCI, VME, SCSI, HIPPI, FibreChannel, and others—can stream data through their XBOWs to and from the memory resident in any node on the system; all concurrent with program data flowing from node to node as requested by multiple IRIX processes running in each CPU in each node.

It is a major triumph of software design that all these facilities can be presented to you, the programmer, under a relatively simple programming model: one program in C or Fortran, with multiple, parallel threads executing the same code. The next chapter examines this programming model.

Chapter 2

SN0 Memory Management

Programming an SN0 system is little different from programming a conventional shared memory multiprocessor such as the Silicon Graphics POWER CHALLENGE R10000. This is, of course, largely because the hardware makes the physically distributed memory work as a shared memory. It is also, however, a result of the IRIX 6.5 operating system, which contains important new capabilities for managing memory and for running the system as efficiently as possible. Although most of these added capabilities are transparent to the user, some new tools are available for fine-tuning performance, and there is some new terminology to go along with them.

This section describes some of the memory-management features of IRIX 6.5, so you can understand what the operating system is trying to accomplish. It familiarizes you with the terminology of memory management, because these terms are used in documenting the compilers and performance tools.

Dealing With Nonuniform Access Time

An SN0 system is a shared memory multiprocessor. It runs the IRIX operating system, which provides a familiar multiuser time-sharing environment as well as familiar compilers, tools, and programming conventions and methods. Even though I/O devices are physically attached to different nodes, any device can be accessed from any program just as you would expect in a UNIX based system. Your old codes run on SN0!

The hardware, however, is different. As described in Chapter 1, “Understanding SN0 Architecture,” there is no longer a central bus into which CPUs, memory, and I/O devices are installed. Instead, memory and peripherals are distributed among the CPUs, and the CPUs are connected to one another via an interconnecting fabric of hub and router chips. This allows unprecedented scalability for a shared memory system. A side effect, however, is that the time it takes for a CPU to access a memory location varies according to the location of the memory relative to the CPU—according to how many hubs and routers the data must pass through. This is *Non-uniform Memory Access (NUMA)*. (For an indication of the access times, see the Read Latency—Max column in Table 1-3 on page 19.)

Although the hardware has been designed so that overall memory access times are greatly reduced compared to earlier systems, and although the variation in times is small, memory access times are, nevertheless, nonuniform, and this introduces a new set of problems to the task of program optimization. These facts are clear:

- A program will run fastest when its data is in the memory closest to the CPU that is executing the code.
- When it is not possible to have all of a program's data physically adjacent to the CPU—for example, if the program uses an array so large it cannot all be allocated in one node—then it is important to find the data that is most frequently used by a CPU, and bring that into nearby memory.
- When a program is executing in parallel on multiple CPUs, the data used by each parallel thread of the algorithm should be close to the CPU that processes the data—even when the parallel threads are processing parallel slices through the same array!

IRIX has been enhanced to take NUMA into account and to make program optimization not only possible, but automatic. The remaining topics of this chapter summarize the NUMA support in IRIX. As a programmer, you are not required to deal with these features directly, but you have the opportunity to take advantage of them using tuning tools described in subsequent chapters.

It is important to remember that the impact of NUMA on performance is, first, important only to programs that incur a lot of secondary cache misses. (Programs that take data mostly out of cache make few references to memory in any location.) Second, these effects apply to multithreaded programs, and to single-threaded programs with large memory requirements. (Single-threaded programs using up to a few tens of megabytes usually run entirely in one node anyway.) Third, NUMA effects are minimal except in larger system configurations. You must have more than 8 CPUs before it is possible to have a path with more than one router in it.

The concepts and terminology are presented anyway, so that you can understand what the operating system is trying to accomplish. In addition, it will make you aware of the programming practices that lead to the most scalable code. (And besides, the technology is fascinating!)

IRIX Memory Locality Management

IRIX 6.5 manages an SN0 system to create a single-system image with the many advanced features required by a modern distributed shared memory operating system. It combines the virtues of the data center environment and the distributed workstation environment, including the following:

- Scalability from 2 to 128 CPUs and from 64 MB to 1 TB of virtual memory
- Upward compatibility with previous versions of IRIX
- Availability features such as IRIS Failsafe, RAID, integrated checkpoint-restart
- Support for interactive and batch scheduling and system partitioning
- Support for parallel-execution languages and tools

Strategies for Memory Locality

The NUMA dependency of memory latency on memory location leads to a new resource for the operating system to manage: *memory locality*. Ideally, each memory access should be satisfied from memory on the same node board as the CPU making the access. This is not always possible; some processes may require more memory than fits on one node board, and different threads in a parallel program, running on separate nodes, may need to access the same memory location. Nevertheless, a high degree of memory locality can be achieved, and the operating system works to maximize memory locality.

For the vast majority of cases, executing an application in the default environment will yield a large fraction of the achievable performance. IRIX obtains optimal memory locality management through the use of a variety of mechanisms described in the following topics.

Topology-aware Memory Allocation

The operating system always attempts to allocate the memory a process uses from the same node on which the process runs. If there is not sufficient memory on the node to allow this, the remainder of the memory is allocated from nodes close to the process.

The usual effect of default allocation policies is to put data close to the CPU that uses it. In some cases of parallel programs, it does not, but the programmer can modify the default policies to suit the program's needs.

Dynamic Page Migration

IRIX can move data from one node to another that is making heavier use of it. IRIX manages memory in virtual pages with a size of 16 KB or larger. (The programmer can control the page size, and in fact can specify different page sizes for data and for code.)

IRIX maintains counters that show which nodes are making the heaviest use of each virtual page. When it discovers that most of the references to a page are coming from a different node, IRIX can move the page to be closer to its primary user. This changes the physical address of the page, but the program uses the virtual address. IRIX copies the data and adjusts the page-translation tables. The movement is transparent to programs. This facility is also called *page migration*.

The programmer can turn page migration on or off for a given program, or turn it on dynamically during specific periods of program execution. These options are discussed in Chapter 8, "Tuning for Parallel Processing."

Replication of Read-Only Pages

IRIX makes extra copies of data pages that are read by multiple nodes—the most important example is the C runtime library code—and spreads the copies throughout the system. It sets the virtual address translation tables for each process to address the copy that is closest to that process. Page replication has several benefits: it reduces the time to access such data, it ensures that the contents of heavily-used pages are not served entirely from a single hub chip, and it reduces the traffic that the interconnecting fabric must carry.

Placing Processes Near Memory

When selecting a CPU on which to initiate a job, the operating system tries to choose a node that, along with its neighbors, provides an ample supply of memory. Thus, should the process's memory requirements grow over time, the additional memory will be allocated from nearby nodes, keeping the access latency low.

Memory Affinity Scheduling

The IRIX scheduler, in juggling the demands of a multiuser environment, makes every attempt to keep each process running on CPUs that are close to the memories in which the process's data resides.

Support for Tuning Options

The IRIX strategies are designed for the "typical" program's needs, and they can be less than optimal for specific programs. Application programmers can influence IRIX memory placement in several ways: through compiler directives; through use of the *dplace* tool (see the dplace(1) reference page, and see Chapter 8, "Tuning for Parallel Processing"); and by runtime calls to special library routines.

Memory Locality Management

The operating system supports memory locality management through a set of low-level system calls. These are not of interest to the application programmer because the capabilities needed to fine-tune performance are available in a high-level tool, *dplace*, and through compiler directives. But a couple of concepts that the system calls rely on are described because terminology derived from them is used by the high-level tools. These concepts are *memory locality domain*s (MLDs) and *policy module*s (PMs).

Every user of an SN0 system implicitly makes use of MLDs and policy modules because the operating system uses them to maximize memory locality. Their use is completely transparent to the user, and they do not need to be understood to use an SN0 system. But for those of you interested in fine-tuning application performance—particularly of large parallel jobs—it can be useful to know that MLDs and policy modules exist, and which types of policies are supported and what they do.

Memory Locality Domain Use

To understand the issues involved in memory locality management, consider the scenario diagrammed in Figure 2-1.

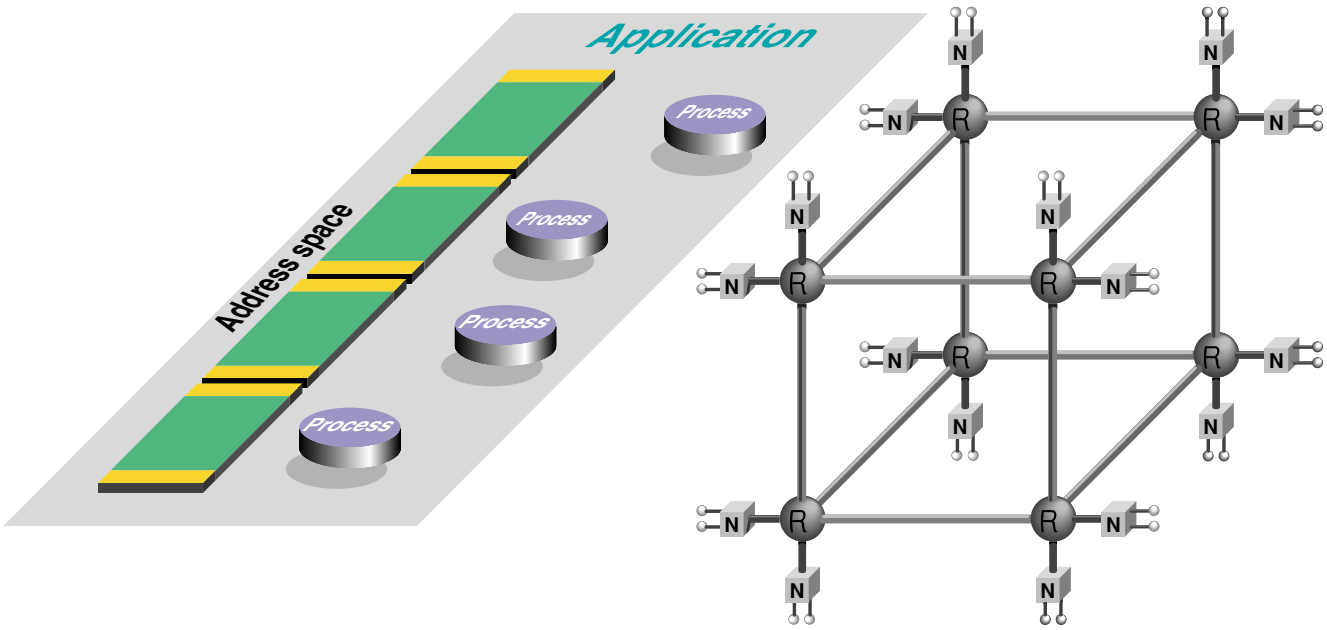
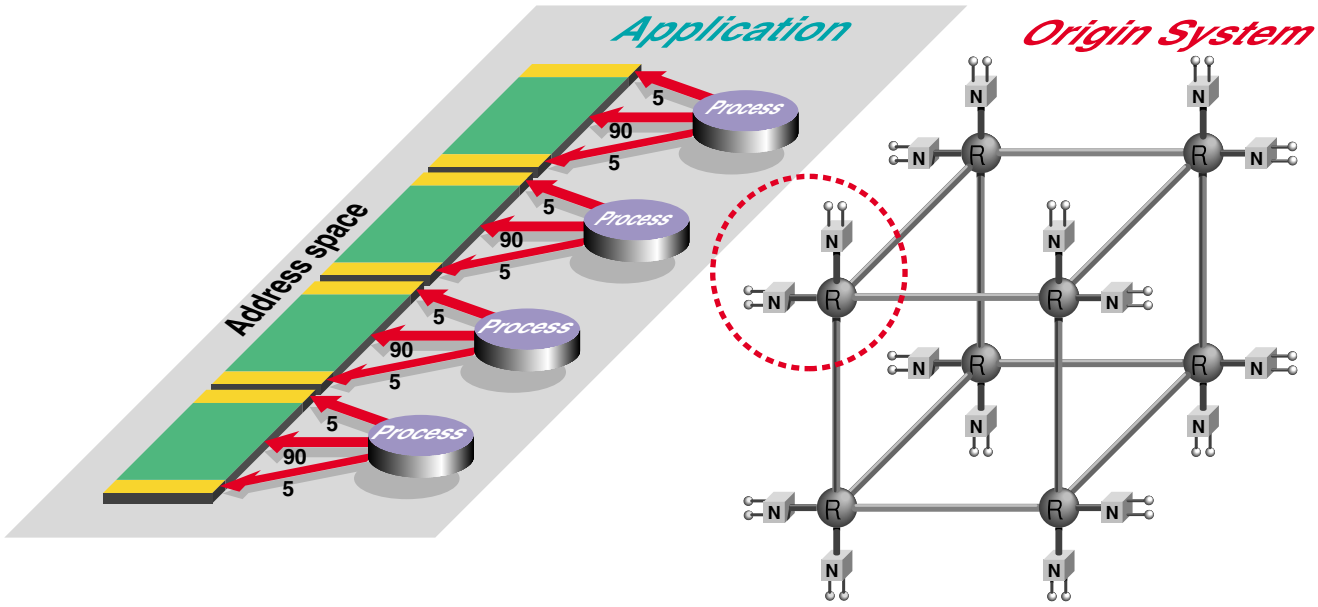


Figure 2-1 Program Address Space Versus SN0 Architecture

Diagrammed on the left in Figure 2-1 is the programmer's view of a shared memory application, consisting of a single virtual address space and four parallel processes. On the right is the architecture of a 32-CPU (16-node) SN0 system. The four application processes can run in any four of the CPUs. The pages that compose the program's address space can be distributed across any combination of one to sixteen nodes. Out of the myriad possible arrangements, how should IRIX locate process execution and data?

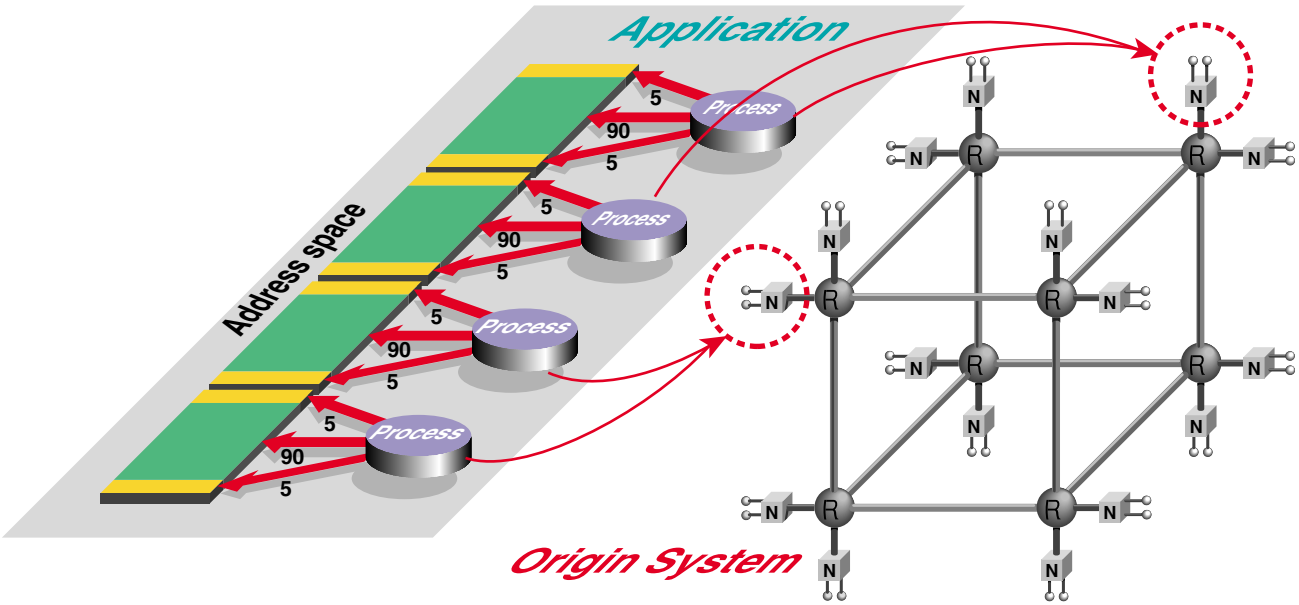
Assume that this application exhibits a relatively simple (and typical) pattern of memory use: each process addresses 5% of its accesses to memory shared with another process; 5% to memory shared with a third process; while 90% of its cache misses are from memory accesses to an almost unshared section of memory. The pattern is best seen graphically, as in Figure 2-2.



Application

Figure 2-2 Parallel Process Memory Access Pattern Versus SN0 Architecture

If IRIX paid little attention to memory locality, the program could end up in the situation shown in Figure 2-3: two processes and half the memory in one corner of the machine, the other processes and memory running in an opposite corner.



Application

Figure 2-3 Parallel Processes at Opposite Corners of SN0 System

The result is acceptable. The first and fourth processes, and 95% of the memory use of the second and third processes, run at local speed. Only when the second and third processes access the data they share, and especially when they update these locations, will data fly back and forth through several routers. The SN0 hardware has been designed to keep the variation in memory latencies small, and accesses to the shared section of memory account for only 5% of two of the processes' cache misses, so this suboptimal placement has a small effect on the performance of the program.

There are situations in which performance could be significantly affected. If absolutely no attention was paid to memory locality, the processes and memory could end up as shown in Figure 2-4.

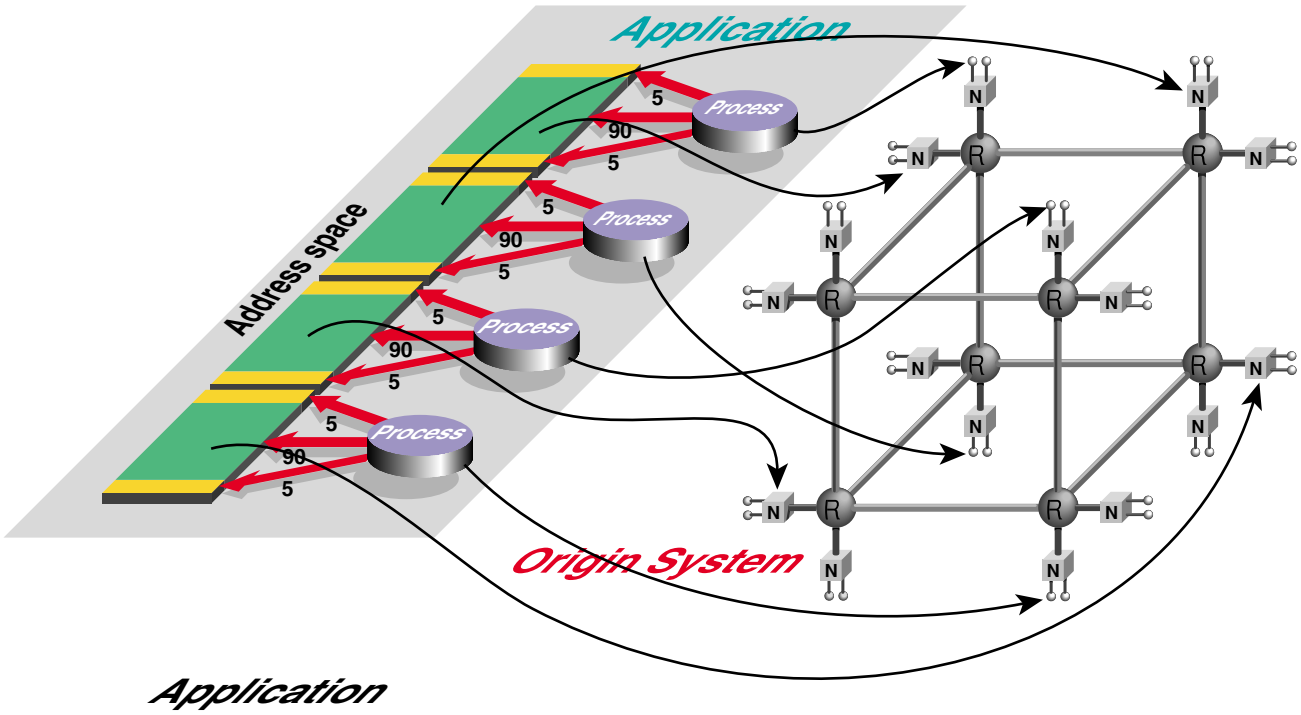


Figure 2-4 Parallel Processes and Memory at Diabolically Bad Locations

Here, each process runs on a different and distant CPU, and the sections of memory it uses is allocated on a different set of distant nodes. In this case, even the accesses to unshared sections of memory—which account for 90% of each process's cache misses—are nonlocal, increasing the costs of accessing memory. In addition, program performance can vary from run to run depending on how close each process ends up to its most-accessed memory. (It is worth mentioning that, even in this least-optimal arrangement, the program would run *correctly*. The poor assignment of nodes would not make the program fail or produce wrong answers, only slow it down and create needless contention for the CrayLink fabric.)

However, the memory locality management mechanisms in IRIX are designed to avoid such situations. Ideally, the processes and memory used by this application are placed in the machine shown in Figure 2-5.

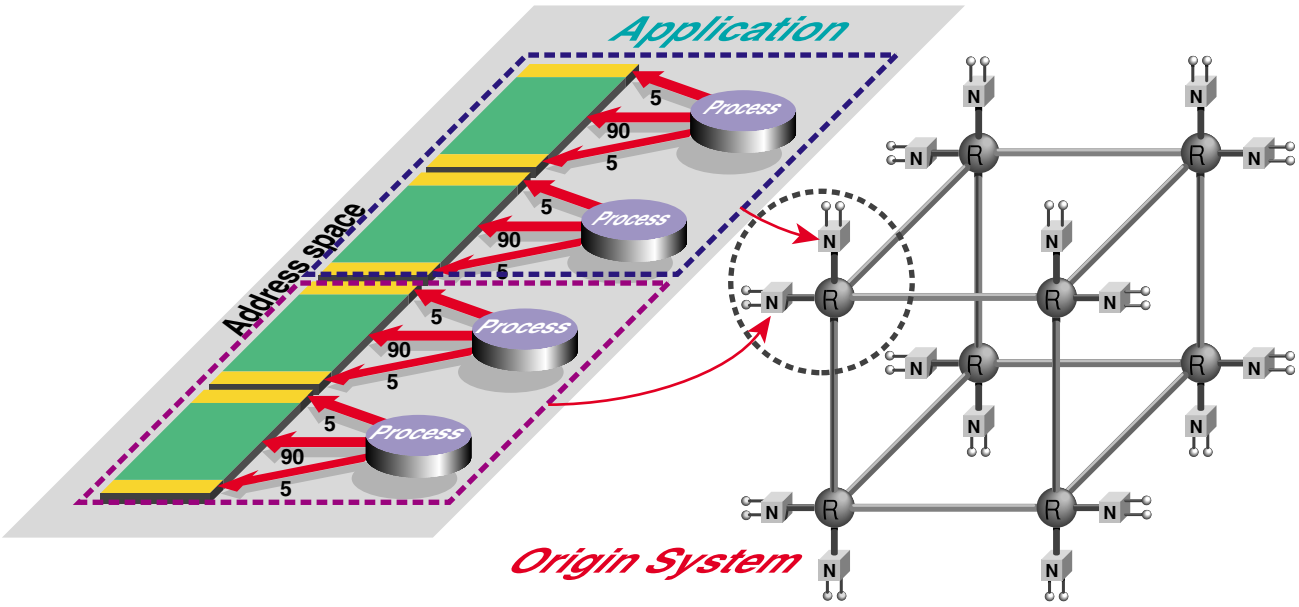


Figure 2-5 Parallel Program Ideally Placed in SN0 System

- The first two processes run on the two CPUs in one node.
- The other two processes run on the CPUs in an adjacent node, one hop away.
- The memory for each pair of processes is allocated in the same node.

To allow IRIX to achieve this ideal placement, two abstractions of physical memory nodes are used:

- Memory locality domains (MLDs)
- Memory locality domain sets (MLDSETs)

A memory locality domain is a source of physical memory. It can represent one node, if there is sufficient memory available for the process(es) that run there, or it can stand for several nodes within a given radius of a center node. For the example application, the operating system creates two MLDs, one for each pair of processes, as diagrammed in Figure 2-6.

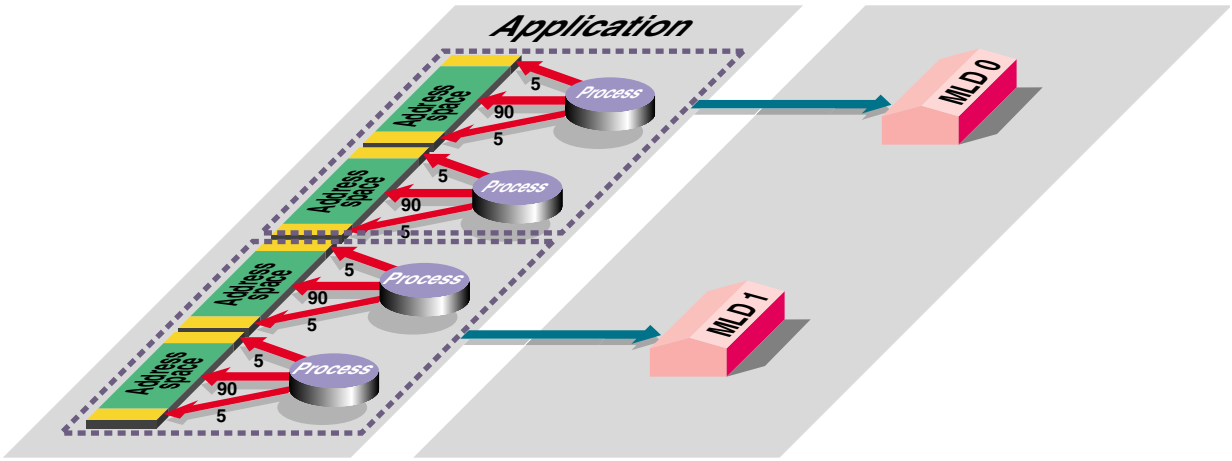


Figure 2-6 Parallel Program Mapped to a Pair of MLDs

It is up to the operating system to decide where in the machine these two MLDs should be placed. Optimal performance requires that they be placed on adjacent nodes, so the operating system needs some additional information.

Memory locality domain sets describe how a program's MLDs should be placed within the machine, and whether they need to be located near any particular hardware devices (for example, close to a graphics pipe). The first property is known as the *topology*, and the second as *resource affinity*.

Several topology choices are available. The default is to let the operating system place the MLDs of the set on a cluster of physical nodes that is as compact as possible. Other topologies allow MLDs to be placed in hypercube configurations (which are proper subsets of the SN0 interconnection topology), or on specific physical nodes. Figure 2-7 shows the MLDs for the example application placed in a one-dimensional hypercube topology with resource affinity for a graphics device.

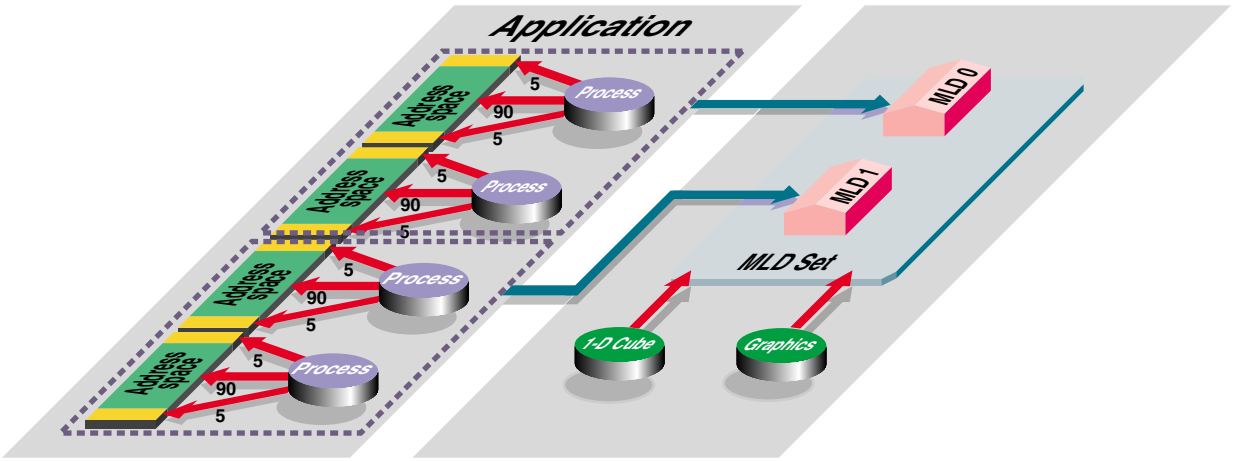


Figure 2-7 Parallel Program Mapped to an MLD Set with Hypercube Topology and Affinity to a Graphics Device

Policy Modules

With the MLDs and MLD set defined, the operating system is almost ready to attach the program's processes to the MLDs, but first policy modules need to be created for the MLDs. Policy modules tell the operating system the following:

- How to place pages of memory in the MLDs.
- Which page size(s) to use.
- What fallback policies to use if the resource limitations prevent the preferred placement and page size choices from being carried out.
- Whether page migration is enabled.
- Whether replication of read-only text is enabled.

The operating system uses a set of default policies unless instructed otherwise. You can change the defaults through the utility *dplace* or via compiler directives. Once the desired policies have been set, the operating system can map processes to MLDs and MLDs to hardware, as shown in Figure 2-8. This ensures that the application threads execute on the nodes from which the memory is allocated.

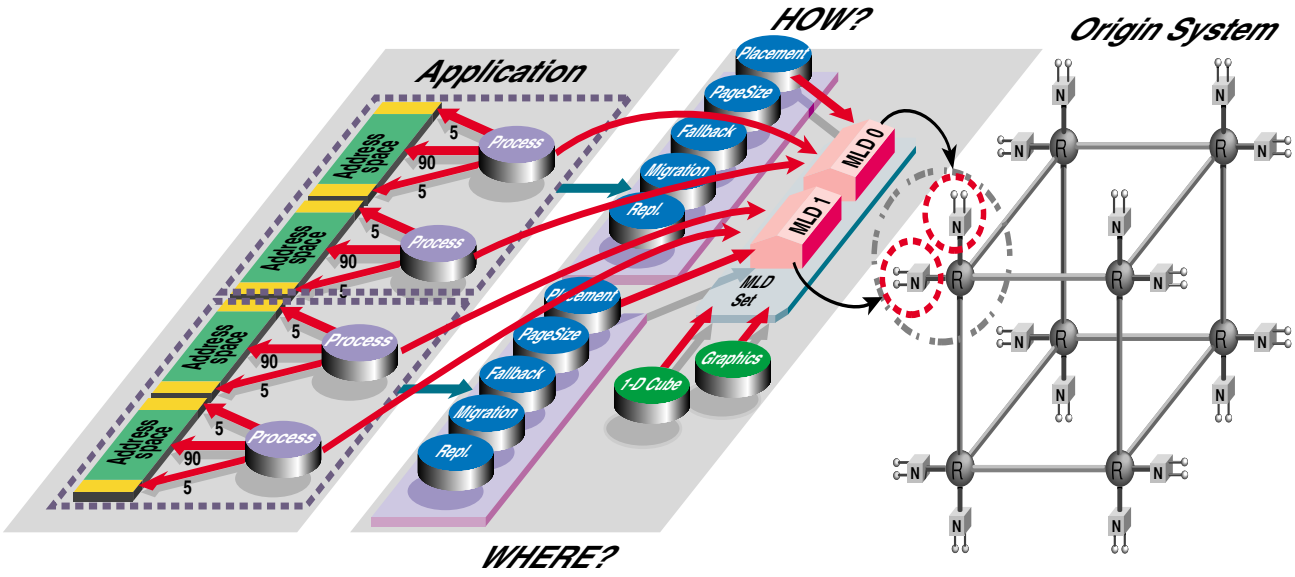


Figure 2-8 Parallel Program Mapped through MLDs to Hardware

Memory Placement for Single-Threaded Programs

The initial placement of data is important for consistently achieving high performance on parallelized applications. It is not an issue for single-threaded programs because they have only one MLD from which to allocate memory.

There is, however, one difference you may see when running a large, single-threaded program on SN0 compared to a bus-based system. When the program has modest memory requirements, it is likely to succeed in allocating all its memory from the node on which it runs. Then all cache misses incur local memory latencies. However, as the program's data requirements grow, it may need to draw memory from nearby nodes. As a result, some cache misses have longer latencies.

Thus the effective time for a cache miss can change, either because the program uses more memory (due either to a change in the algorithm or to larger input data), or because other jobs consume more of the memory on the node. Note that if the program is allocating significantly more memory, it is likely to run longer in any case, because it is

doing more work. Any NUMA variation may be unnoticeable beside the algorithmic difference.

Data Placement Policies

Although it may be obvious to the you, the application programmer, what the program's memory access patterns are, there is no way for the operating system to know this. The placement policies determine how the memory for virtual pages is allocated from the different MLDs in the MLD set. Three memory placement policies are available:

1. First-touch, in which memory is allocated from the MLD associated with the first process to access the page (in other words, the process that first faults that page).
2. Fixed, in which memory is allocated from a specific MLD.
3. Round-robin, in which pages of memory are allocated in a round-robin fashion from each of the MLDs in the MLD Set in turn.

Using First-Touch Placement

The default policy is first-touch. It works well with single-threaded programs, because it keeps memory close to the program's one process. It also works well for programs that have been parallelized completely, so that each parallel thread allocates and initializes the memory it uses.

When the example program above (Figure 2-8) is running, each CPU accesses three segments of data: a 90% segment accessed by no other node and two 5% pieces, each of which is also accessed by a neighboring CPU. In the ideal data layout, the 90% piece for each CPU is stored in its local memory. In addition, the two adjacent 5% pieces are stored locally, while the two remaining 5% pieces are stored in the neighboring CPUs. This distributes responsibility for storing the data equally among all CPUs and ensures that they all incur the same cost for accessing nonlocal memory.

If the initialization of memory is done in parallel in each process, each CPU is the first to touch its 90% piece and the two adjacent 5% pieces, causing those segments to be allocated locally. This is exactly where we want these segments to reside. When you program memory initialization in each program thread independently, the first-touch policy guarantees good data placement. The common alternative, to initialize all of memory from a single master process before starting the parallel processes, has just the wrong effect, placing all memory into a single node.

Although this extra consideration is not required in a bus-based system such as POWER CHALLENGE, it is simple to do and does not require you to learn anything new, such as the compiler directives, which also allow data to be placed optimally.

If a program has not been completely parallelized, the first-touch policy may not be the best one to use. For the example application, if all the data are initialized by just one of the four processes, all the data will be allocated from a single MLD, rather than being evenly spread out among the four MLDs. This introduces two problems:

- Accesses that should be local are now remote.
- All the processes' cache misses are satisfied from the same memory. This can cause a performance bottleneck by overloading the hub in that node.

Even in this case, if dynamic page migration is enabled, the data will eventually move to where it is used, so ultimately pages should be placed correctly. This works well for applications in which there is one optimal data placement, and the application runs long enough (minutes) for the data to migrate to their optimal locations.

Using Round-Robin Placement

In some complicated applications, different placements of data are needed in different phases of the program, and the program phases alternate so quickly, that there is not time for the operating system to migrate the data to the best location for one program phase before that phase ends and a new one begins.

For such applications, a difficult solution is to perform explicit page migration using programmed compiler directives. A simple solution is to use the round-robin placement policy. Under this policy, memory is evenly distributed, by pages, among the MLDs. Any one page is not likely to be in an optimal location at a particular phase of the program; however, by spreading memory accesses across all the nodes where the program runs, you avoid creating performance bottlenecks.

Avoiding bottlenecks is actually a more important consideration than getting the lowest latency. The variation in memory latency is moderate, but if all the data are stored in one node, the outgoing bandwidth of that one hub chip is divided among all the other CPUs that use the data.

Using Fixed Placement

The final placement policy is fixed placement, which places pages of memory in a specific MLD. You invoke this policy using the compiler placement directives. These are a

convenient way to specify the optimal placement of data, when you are sure you know what it is. You can specify different placements in different parts of a program, so these directives are ideal for complicated applications that need different data placements during different processing phases.

Achieving Good Performance in a NUMA System

The memory locality management automatically performed by IRIX means that most SN0 programs can achieve good performance and enhanced scalability without having to program any differently from the way they do on any other system. But some programs that were developed on a uniform memory access architecture might run more slowly than they could. This section summarizes when you can expect to see the effects of the NUMA architecture and what tools you can use to minimize them.

Single-Threaded Programs under NUMA

Even though SN0 is a highly parallel system, the vast majority of programs it runs use only a single CPU. Nothing new or different needs to be done to achieve good performance on such applications. Tune them just as you would for any other system. Chapter 3, "Tuning for a Single Process," provides a detailed discussion of the tools available and steps you can take to improve the performance of single-threaded programs.

There is one new capability in the SN0 architecture that single-CPU programs can sometimes use to improve performance, namely, support for multiple page sizes. Recall that page size is one of the policies that IRIX 6.5 can apply in allocating memory for processes. Normally, the default page size of 16 KB is used, but for programs that incur a performance penalty from a large number of virtual page faults, it can be beneficial to use a larger page size. For single-CPU programs, page size is controlled via *dplace* and explained under "Using Larger Page Sizes to Reduce TLB Misses" on page 147.

Parallel Programs under NUMA

Single-CPU tuning accounts for most of the performance work you do. Don't get lazy here: using the proper compiler flags and coding practices can yield big improvements in program performance. Once you have made your best effort at single-CPU tuning, you can turn your attention to issues related to parallel programming. Parallel programs are separate into two classes:

- MP library programs. These use the shared memory parallel programming directives that have been available in Silicon Graphics compilers for many years. It also includes programs using MPI version 3 or later, as distributed with IRIX.
- Non-MP library programs. These include programs using the IRIX facilities for starting parallel processes, programs using the pthreads model, programs using the PVM library, and programs using the Cray shared-memory models.

For programs in the second class, it is a good idea to use *dplace* to specify a memory placement policy. This is covered in Chapter 8, "Tuning for Parallel Processing."

Programs in the first class that make good use of the CPU's caches see little effect from NUMA architecture. The reason for this is that memory performance is affected by memory latencies only if the program spends a noticeable amount of time actually accessing memory, as opposed to cache. Cache-friendly programs have very few cache misses, so the vast majority of their data accesses are satisfied by the caches, not memory. Thus the only NUMA effect these programs see is scalability to a larger number of CPUs.

The *perfex* and Speedshop tools can be used to determine if a program is cache-friendly and, if not, where the problems are. You can find a detailed discussion in Chapter 3, "Tuning for a Single Process."

Not all programs can be made cache friendly, however. That only means these memory-intensive programs will spend more time accessing memory than their cache-friendly brethren, and in general will run at lower performance. This is just one of the facts of life for cache-based systems, which is to say, all computer systems except very expensive vector CPUs.

When there is a performance or scalability problem, the data may not have been well placed, and modifying the placement policies can fix this. First, try a round-robin placement policy. (For programs that have been parallelized using the Silicon Graphics MP directives, round-robin placement can be enabled by changing only an environment variable (see "Trying Round-Robin Placement" on page 207). If this solves the performance problem, you're done. If not, try enabling dynamic page migration (done with yet another environment variable, see "Trying Dynamic Page Migration" on page 209). Both can be tried in combination.

Often, these policy changes are all that is needed to fix a performance problem. They are convenient because they require no modifications to your program. When they do not fix a performance or scalability problem, you need to modify the program to ensure that the data are optimally placed. This can be done in a couple of ways.

You can use the default first-touch policy, and program so that the data are first accessed by the CPU in whose memory they should reside. This programming style is easy to use, and it does not require learning new compiler directives. (See "Programming For First-Touch Placement" on page 216.)

The second way to ensure proper data layout is to use the data placement directives, which permit you to specify the precise data layout that is optimal for your program. The use of these directives is described in "Using Data Distribution Directives" on page 222.

Summary

While the distributed shared memory architecture of the SN0 systems introduces new complexities, most of them are handled automatically by IRIX, which uses a number of internal strategies to ensure that processes and their data end up reasonably close together. Normally, single-threaded programs operate very well, showing no effects from the NUMA architecture, other than a run-to-run variation in memory latency that depends on the program's dynamic memory use.

When programs are made parallel using the SGI-supplied compiler and MP library, and when parallel programs are based on the SGI-supplied MPI version 3 library, they use appropriate MLDs automatically and are likely to run well without further tuning for NUMA effects. Parallel programs based on other models may gain from programmer control over memory placement, and the tools for this control are available. All these points are examined in detail in Chapter 8, "Tuning for Parallel Processing."

Chapter 3

Tuning for a Single Process

Now that you have been introduced to the SN0 architecture, it is time to address "tuning," that is, how to make your programs run their fastest on this architecture. One answer, in the SN0 environment, is to make the program use multiple CPUs in parallel. However, this should be the last step. First, make your program run as efficiently as a possible as a single process.

The process of tuning can be divided into the following steps:

1. Make sure the program produces the right answers. This is especially important when first porting a program to IRIX 6.5, 64-bit computing, and SN0.
2. Use existing, tuned library code.

   Steps 1 and 2 are covered in this chapter.
3. Profile the program's execution and analyze its dynamic behavior. The many tools for this purpose are covered in Chapter 4, "Profiling and Analyzing Program Behavior."
4. Make the compiler produce the most efficient code. The use of the many interrelated compiler options is covered in Chapter 5, "Using Basic Compiler Optimizations."
5. Modify the program to access memory efficiently. This is covered in Chapter 6, "Optimizing Cache Utilization."
6. Exploit the compiler's ability to optimize loops. Loop optimization is covered in Chapter 7, "Using Loop Nest Optimization."

When you have done all these things, and the program still takes too long, Chapter 8, "Tuning for Parallel Processing," covers parallelization.

Getting the Right Answers

You can make any program run as fast as you want—if you don't insist on correct results. However, the very first rule in tuning is to make sure that the program generates the correct results. Although this may seem obvious, it is easy to forget to check the answers along the way as you make performance improvements. A lot of frustration can be avoided by making only one change at a time, verifying that the program is correct after each change.

In addition to problems introduced by performance changes, correctness issues may be raised merely by porting a code from one system to another, or by compiling to a different environment. Most programs that are written in a high-level language and adhere to the language standard will port with no difficulties. This is particularly true for standard-compliant Fortran-77 programs. Some C programs can cause problems.

Selecting an ABI and ISA

IRIX 6.5 is a 64-bit operating system. Although programs can be (and often are) compiled and run in a 32-bit address space, you may need to take advantage of the larger address space and so compile in the 64-bit Application Binary Interface (ABI). (For a survey of the available ABIs and the differences among them, see the ABI(5) reference page.) All modules of a program must use the same ABI.

When a program is written in a portable manner, you can compile it for a different ABI merely by specifying one of the compiler flags *-32*, *-n32*, or *-64*. However, a change of ABI can cause a program to fail, because of a variety of subtle problems. All of these issues are discussed in depth in the *MIPSpro 64-Bit Porting and Transition Guide* listed in "Related Documents" on page xxix.

Old 32-Bit ABI

The default ABI is the "old" 32-bit environment (compiler flag *-32*). It is the default because it provides the greatest backward compatibility to earlier versions of IRIX: it was the only ABI before IRIX version 6. Programs compiled *-32* are limited to the MIPS I and MIPS II instruction sets, on the assumption that they must be able to execute in an older computer. If you want to use the capabilities of the MIPS III or MIPS IV instructions sets—and you do, if performance is a goal—you need to compile to a different ABI.

New 32-Bit ABI

With the "new" 32-bit ABI (compiler flag *-n32*), the program uses a 32-bit address space. Pointers and long integers are 32 bits in size, so there are fewer portability issues (for example, a binary file containing 32-bit integers written by an old program can be mapped by the identical header file in an *-n32* program).

The advantage of *-n32* is that it allows programs use either the MIPS III or the MIPS IV ISAs (selected by the *-mips3* and *-mips4* compiler flags). These ISAs use more and longer registers, and use a faster protocol for subroutine calls, than the old 32-bit ABI did.

64-Bit ABI

When compiled with the *-64* flag, the program runs in a 64-bit address space. This permits definition of immense arrays of data. Pointers and long integers are 64 bits in length. Other than that, *-64* and *-n32* are essentially the same. That is, there is no performance advantage to the use of *-64*, and indeed, because pointers take up more memory, a program that has many pointers may run more slowly when compiled *-64*. Use it only when your program requires memory data totalling more than 2 Gigabytes in size.

Specifying the ABI

Unless you require your program to run on a MIPS R4000 system (such as an Indy workstation), use one of the combinations *-n32 -mips4* or *-64 -mips4*. If backward compatibility with the R4x00 CPU is needed, use *-n32 mips3*.

On all non-R8000 systems, the default execution environment is *-32* and *-mips2*. On CHALLENGE and Onyx R8000-based systems, these defaults are *-64 -mips4*. Because the defaults vary by system type, and because you may compile on a workstation for execution on a server (or vice versa), it is wise to specify the desired ABI and ISA explicitly in every compile. The best way to do this is with a makefile, as described under "Using a Makefile" on page 92.

You can also set the environment variable SGI_ABI to -32, -n32, or -64 to specify a default ABI.

Dealing with Porting Issues

A program can fail to port to IRIX for many reasons. Different compilers handle standards violations differently, so nonstandard programs often generate different results when rehosted.

Uninitialized Variables

A common example is a Fortran program that assumes that all variables are initialized to zero. While this initialization occurred on most older machines, it is not common on newer, stack-based architectures. When a program making such an assumption is ported to IRIX, it can generate incorrect results.

To detect this error early, you can compile with the *-trapuv* flag, which initializes all uninitialized variables in a program with the value 0xFFFA5A5A. When this value is used as a floating point variable, it is seen as a floating point NaN ("not a number") and causes an immediate floating point trap. When it is used as a pointer, an address or segmentation violation generally occurs. By running the program with a debugger, the use of an uninitialized variable will quickly be obvious.

Although fixing the program is the "right thing to do," it does require some effort. An alternative that requires almost no effort is to use the *-static* flag, which causes all variables to be allocated from a static data area rather than the stack. Static variables are initialized to zero. There is a penalty to the easy way out: use of the *-static* flag hurts program performance; a 20 per cent penalty is common.

Computational Differences

Other types of porting failures can occur as well. The results generated on one computer may disagree with those from another because of differences in the floating point format, or because of differences in the generated code that cause the roundoff error to change. You need to understand the particular program to determine if these differences are significant.

In addition, some programs may even fail to compile on a new system if extensions are not supported or if they are provided through a different syntax. Fortran pointers provide a good example of this; there is no pointer type in the Fortran standard, and different systems provide this extension in incompatible ways. In such situations, the code must be modified to be compatible with the new compiler. Similarly, if a program makes calls to a particular vendor's libraries, these calls must be replaced with equivalent entry points from the new vendor's libraries.

To verify the existence and syntax of Fortran features in Silicon Graphics compilers, see the programmer's guides listed in "Related Documents" on page xxix.

Finally, programs sometimes have mistakes in them that by pure luck have not yet caused problems but that, when compiled on a new system, cause errors to appear. Writing beyond the end of an array can show this behavior: whether storing past the end of the array causes a problem depends on how the compiler lays out variables in memory.

For Fortran, use the *-check_bounds* flag to instruct the compiler to generate run-time subscript range checking. If an out-of-bounds array reference is made, the program aborts and displays an error message indicating the illegal reference and the line of code where it occurred.

Exploiting Existing Tuned Code

The quickest and easiest way to improve a program's performance is to link it with libraries already tuned for the target hardware. The standard math library is so tuned, and there are optional libraries that can provide performance benefits: *libfastm*, *CHALLENGEcomplib*, and *SCSL*.

Standard Math Library

For the standard libraries such as *libc*, hardware-specific versions are automatically linked, based on the compiler's information about the target system. The compiler assumes that the current system is the target for execution, but you can tell it otherwise with compiler options. The *-TARG* option group (described in the cc(1) or f77(1) reference pages) describes the target system by CPU type or by Silicon Graphics processor board type ("IP" number). Alternatively, you can specify the *-r10000*, *-r8000*, or *-r5000* flags to specify a particular MIPS CPU.

The standard math library includes special "vector intrinsics," that is, vectorized versions of the functions **vacos**, **vasin**, **vatan**, **vcos**, **vexp**, **vlog**, **vsin**, **vsqrt**, and **vtan** (plus single-precision versions whose names are made by appending an **f**). These functions are designed to take maximum advantage of the pipelining characteristics of the CPU when processing a vector of numbers.

You can write explicit calls to the vector functions, passing input and output vectors. In certain circumstances, the loop-nest optimizer of the Fortran compilers recognizes a vector loop and automatically replaces it with a vector intrinsic call (see the Fortran Programmer's Guides listed under "Related Documents" on page xxix).

The standard math library is described in the math(3) reference page. This library is linked in automatically by Fortran. When you link using the *ld* or *cc* commands, use the *-lm* flag.

libfastm Library

The *libfastm* library provides highly optimized versions of a subset of the routines found in the standard *libm* math library. Optimized scalar versions of **sin**, **cos**, **tan**, **exp**, **log**, and **pow** for both single and double precision are included. To link *libfastm*, append *-lfastm* to the end of the link line: `cc -o` *modules...* `-lfastm`.

Separate versions of *libfastm* exist for the R10000, R8000, and R5000. When the target system is not the one used for compiling, explicitly specify the target CPU. For details, see the libfastm(3) reference page.

CHALLENGEcomplib Library

CHALLENGEcomplib is a library of mathematical routines that carries out linear algebra operations, Fast Fourier Transforms (FFTs), and convolutions. It contains implementations of the well-known public domain libraries LINPACK, EISPACK, LAPACK, FFTPACK, and the Level-1, -2, and -3 BLAS, but all tuned specifically for high performance on the MIPS IV ISA. In addition, the library contains proprietary sparse solvers, also tuned to the hardware.

Many *CHALLENGEcomplib* routines have been parallelized, so as to use multiple CPUs concurrently to shorten solution time. A parallelized algorithm runs in a one-CPU system (or when a program is confined to a single CPU), but it incurs additional overhead. *CHALLENGEcomplib* comes in both sequential and parallel versions. To link the sequential version, add *-lcomplib.sgimath* to the link line:

```
f77 -o modules -lcomplib.sgimath -lfastm
```

To link the parallel version, use *-lcomplib.sgimath_mp*; in addition, the *-mp* flag must be used when linking in the parallel code:

```
f77 -mp -o modules -lcomplib.sgimath_mp -lfastm
```

SCSL Library

SCSL, the Cray Scientific Library, is an optimized library that will eventually replace both *CHALLENGEcomplib* and Cray's *libsci* library. *SCSL* version 1.0 contains FFT routines, Level-1, -2, and -3 BLAS, and LAPACK routines, many of them parallelized. *SCSL* is distributed as a separate product from IRIX, but is available at no charge to IRIX users. *SCSL* ships automatically to all SN0 systems.

Note that the user interface for the FFT routines is different from the interface used in *CHALLENGEcomplib*. Some extended BLAS routines, which are available in *libsci* but not in *CHALLENGEcomplib*, have been included in SCSL.

You link *SCSL* into your program by using *-lscs* for the single-threaded version, or *-lscs_mp* for the parallelized version of the library.

Tip: Both *CHALLENGEcomplib* and *SCSL* define names also found in *libfastm*, so if you want to call *libfastm* it should be named last in the link command.

Summary

Many working UNIX applications compile and run correctly without modification in IRIX. Nevertheless, dependency on vendor-specific interfaces, or on compiler behavior, or incorrect assumptions about data representation, can cause errors, and the first task is to make sure the program processes its full range of input to produce correct answers. Then, if the program uses external math functions, link it with a tuned library and again make sure that correct answers emerge. Then the program is ready to be tuned.

Chapter 4

Profiling and Analyzing Program Behavior

When confronted with a program composed of hundreds of modules and thousands of lines of code, it would require a heroic (and inefficient) effort to tune the entire program. Tuning should be concentrated on those few sections of the code where the work will pay off with the biggest gains in performance. These sections of code are identified with the help of the profiling tools.

Profiling Tools

The hardware counters in the R10000 CPU, together with support from the IRIX kernel, make it possible to record the run-time behavior of a program in many ways without modifying the code. The profiling tools are summarized as follows:

- *perfex* runs a program and reports exact counts of any two selected events from the R10000 hardware event counters. Alternatively, it time-multiplexes all 32 countable events and reports extrapolated totals of each. *perfex* is useful for identifying what problem (for example, secondary data cache misses) is hurting the performance of your program the most.
- Speedshop (actually, the *ssrun* command) runs your program while sampling the state of the program counter and stack, and writes the sample data to a file for later analysis. You can choose from a wide variety of sampling methods, called "experiments." Speedshop is useful for locating where in your program the performance problems occur.
- *prof* analyzes a Speedshop data file and displays it in a variety of formats.
- *dprof*, like Speedshop, samples a program while it is executing, and records the program's memory access information as a histogram file. It identifies which data structures in the program are involved in performance problems.
- *dlook* runs a program and at the end, displays the placement of its memory pages to the standard error file.

Use these tools to find out what constrains the program and which parts of it consume the most time. Through the use of a combination of these tools, you can identify most performance problems.

Analyzing Performance with Perfex

The simplest profiling tool is *perfex*, documented in the perfex(1) reference page. It is conceptually similar to the familiar *timex* program, in that it runs a subject program and records data about the run:

```
% perfex options command arguments
```

The subject command and its arguments are given. *perfex* sets up the IRIX kernel interface to the R10000 hardware event counters (for a detailed overview of the counters, see Appendix B, "R10000 Counter Event Types"), and forks the subject program. When the program ends, *perfex* writes counter data to standard output. Depending on the options you specify, *perfex* reports an exact count of one or two countable events (see Table B-1 on page 274 for a list), or it reports an approximate count of all 32 countable event types. *perfex* gathers the information with no modifications to the subject program, and with only a small effect on its execution time.

Taking Absolute Counts of One or Two Events

Use *perfex* options to specify one or two events to be counted. When you do this, the counts are absolute and repeatable. You can experiment with *perfex* by applying it to familiar commands, as in Example 4-1.

Example 4-1 Experimenting with perfex

```
> perfex -e 15 -e 18 date +%w.%V
3.26
Summary for execution of date +%w.%V
15 Graduated instructions.....................................      41066
18 Graduated loads............................................       9021
```

perfex runs the subject program and reports the exact counts of the requested events. You can use this mode to explore any program's behavior. For example, you could run a program, counting graduated instructions and graduated floating-point instructions (`perfex -e 15 -e 21`), under a range of input sizes. From the results, you could draw a graph showing how instruction count grows as an exact function of input size.

Taking Statistical Counts of All Events

When you specify option *-a* (all events), *perfex* multiplexes all 32 events over the program run. The IRIX kernel rotates the counters each time the dispatching clock ticks (100 HZ). Each event count is active 1/16 of the time during the program run, and then scaled by 16 in the report.

Because they are based on sampling, the resulting counts have some unavoidable statistical error. When the subject program runs in a stable execution mode for a number of seconds, the error is small and the counts are repeatable. When the program runs only a short time, or when it shifts rapidly between radically different regimes of instruction or data use, the counts are less dependable and less repeatable.

Example 4-2 shows the *perfex* command and output, applied to a CPU-intensive sample program called *adi2.f* (see Example C-1 on page 288).

Example 4-2 Output of perfex -a

```
% perfex -a -x adi2
WARNING: Multiplexing events to project totals--inaccuracy possible.
 Time:      7.990 seconds
 Checksum:  5.6160428338E+06
 0 Cycles.....................................................   1645481936
 1 Issued instructions........................................    677976352
 2 Issued loads...............................................    111412576
 3 Issued stores..............................................     45085648
 4 Issued store conditionals..................................            0
 5 Failed store conditionals..................................            0
 6 Decoded branches...........................................     52196528
 7 Quadwords written back from scache.........................     61794304
 8 Correctable scache data array ECC errors...................            0
 9 Primary instruction cache misses...........................         8560
10 Secondary instruction cache misses.........................          304
11 Instruction misprediction from scache way prediction table..          272
12 External interventions.....................................         6144
13 External invalidations.....................................        10032
14 Virtual coherency conditions...............................            0
15 Graduated instructions.....................................    371427616
16 Cycles.....................................................   1645481936
17 Graduated instructions.....................................    400535904
18 Graduated loads............................................     90474112
19 Graduated stores...........................................     34776112
20 Graduated store conditionals...............................            0
21 Graduated floating point instructions......................     28292480
```

```
22 Quadwords written back from primary data cache..............     32386400
23 TLB misses..................................................      5687456
24 Mispredicted branches.......................................       410064
25 Primary data cache misses...................................     16330160
26 Secondary data cache misses.................................      7708944
27 Data misprediction from scache way prediction table.........       663648
28 External intervention hits in scache........................         6144
29 External invalidation hits in scache........................         6864
30 Store/prefetch exclusive to clean block in scache...........      7582256
31 Store/prefetch exclusive to shared block in scache..........         8144
```

The *-x* option requests that *perfex* also gather counts for kernel code that handles exceptions, so the work done by the OS to handle TLB misses is included in these counts.

Getting Analytic Output with the -y Option

The raw event counts are interesting, but it is more useful to convert them to elapsed time. Some time estimates are simple; for example, dividing the cycle count by the machine clock rate gives the elapsed run time (1645481936 ÷ 195 MHz = 8.44 seconds). Other events are not as simple and can be stated only in terms of a range of times. For example, the time to handle a primary cache miss varies depending on whether the needed data are in the secondary cache, in memory, or in the cache of another CPU. You can request analysis of this kind using the *-y* option.

When you use both *-a* and *-y, perfex* collects and displays all event counts, but it also displays a report of estimated times based on the counts. Example 4-3 shows, again, the program *adi2.f*.

Example 4-3 Output of perfex -a -y

```
% perfex -a -x -y adi2
WARNING: Multiplexing events to project totals--inaccuracy possible.
 Time:      7.996 seconds
 Checksum:  5.6160428338E+06
                                                                  Based on 196 MHz IP27
                                                                                Typical      Minimum      Maximum
   Event Counter Name                                          Counter Value   Time (sec)   Time (sec)   Time (sec)
=============================================================================================================================
 0 Cycles.....................................................   1639802080     8.366337     8.366337     8.366337
16 Cycles.....................................................   1639802080     8.366337     8.366337     8.366337
26 Secondary data cache misses................................      7736432     2.920580     1.909429     3.248837
23 TLB misses.................................................      5693808     1.978017     1.978017     1.978017
 7 Quadwords written back from scache.........................     61712384     1.973562     1.305834     1.973562
25 Primary data cache misses..................................     16368384     0.752445     0.235504     0.752445
```

```
22 Quadwords written back from primary data cache.....     32385280     0.636139     0.518825     0.735278
 2 Issued loads.....                                      109918560     0.560809     0.560809     0.560809
18 Graduated loads.....                                    88890736     0.453524     0.453524     0.453524
 6 Decoded branches.....                                   52497360     0.267844     0.267844     0.267844
 3 Issued stores.....                                      43923616     0.224100     0.224100     0.224100
19 Graduated stores.....                                   33430240     0.170562     0.170562     0.170562
21 Graduated floating point instructions.....              28371152     0.144751     0.072375     7.527040
30 Store/prefetch exclusive to clean block in scache.....   7545984     0.038500     0.038500     0.038500
24 Mispredicted branches.....                               417440     0.003024     0.001363     0.011118
 9 Primary instruction cache misses.....                     8272     0.000761     0.000238     0.000761
10 Secondary instruction cache misses.....                    768     0.000290     0.000190     0.000323
31 Store/prefetch exclusive to shared block in scache.....  15168     0.000077     0.000077     0.000077
 1 Issued instructions.....                                673476960     0.000000     0.000000     3.436107
 4 Issued store conditionals.....                                0     0.000000     0.000000     0.000000
 5 Failed store conditionals.....                                0     0.000000     0.000000     0.000000
 8 Correctable scache data array ECC errors.....                 0     0.000000     0.000000     0.000000
11 Instruction misprediction from scache way prediction table..   432     0.000000     0.000000     0.000002
12 External interventions.....                                6288     0.000000     0.000000     0.000000
13 External invalidations.....                                9360     0.000000     0.000000     0.000000
14 Virtual coherency conditions.....                             0     0.000000     0.000000     0.000000
15 Graduated instructions.....                            364303776     0.000000     0.000000     1.858693
17 Graduated instructions.....                            392675440     0.000000     0.000000     2.003446
20 Graduated store conditionals.....                             0     0.000000     0.000000     0.000000
27 Data misprediction from scache way prediction table.....  679120     0.000000     0.000000     0.003465
28 External intervention hits in scache.....                  6288     0.000000     0.000000     0.000000
29 External invalidation hits in scache.....                  5952     0.000000     0.000000     0.000000
Statistics
--------------------------------------------------------------------------------
Graduated instructions/cycle.....                            0.222163
Graduated floating point instructions/cycle.....             0.017302
Graduated loads & stores/cycle.....                          0.074595
Graduated loads & stores/floating point instruction.....     5.422486
Mispredicted branches/Decoded branches.....                  0.007952
Graduated loads/Issued loads.....                            0.808696
Graduated stores/Issued stores.....                          0.761099
Data mispredict/Data scache hits.....                        0.078675
Instruction mispredict/Instruction scache hits.....          0.057569
L1 Cache Line Reuse.....                                     6.473003
L2 Cache Line Reuse.....                                     1.115754
L1 Data Cache Hit Rate.....                                  0.866185
L2 Data Cache Hit Rate.....                                  0.527355
Time accessing memory/Total time.....                        0.750045
L1--L2 bandwidth used (MB/s, average per process).....     124.541093
Memory bandwidth used (MB/s, average per process).....     236.383187
MFLOPS (average per process).....                            3.391108
```

Interpreting Maximum and Typical Estimates

For each count, the "maximum," "minimum," and "typical" time cost estimates are reported. Each is obtained by consulting an internal table that holds the maximum, minimum, and typical costs for each event, and multiplying this cost by the count for the event. Event costs are usually measured in terms of machine cycles, so the cost of an event depends on the clock speed of the CPU, which is also reported in the output.

The "maximum" value in the table corresponds to the worst-case cost of a single occurrence of the event. Sometimes this can be a pessimistic estimate. For example, the maximum cost for graduated floating-point instructions assumes that every floating-point instruction is a double-precision reciprocal square root, which is the most costly R10000 floating-point instruction.

Because of the latency-hiding capabilities of the R10000 CPU, most events can be overlapped with other operations. As a result, the minimum cost of virtually any event could be zero. To avoid simply reporting minimum costs of zero, which would be of no practical use, the minimum time reported corresponds to the best-case cost of a single occurrence of the event. The best-case cost is obtained by running the maximum number of simultaneous occurrences of that event and averaging the cost. For example, two floating-point instructions can complete per cycle, so the best case cost is 0.5 cycles per floating-point instruction.

The typical cost falls somewhere between minimum and maximum and is meant to correspond to the cost you see in average programs.

The report shows the event counts and cost estimates sorted from most costly to least costly, so that the events that are most significant to program run time appear first. This is not a true profile of the program's execution, because the counts are approximate and because the event costs are only estimates. Furthermore, because events do overlap one another, the sum of the estimated times normally exceeds the program's true run time. This report should be used only to identify which events are responsible for significant portions of the program's run time, and to get an estimate of relative costs.

In Example 4-3, the program spends a significant fraction of its time handling secondary cache and TLB misses (together, as much as 5 seconds of its 8.4-second run time). Tuning measures focussed on these areas could have a significant effect on performance.

Interpreting Statistical Metrics

In addition to the event counts and cost estimates, *perfex -y* also reports a number of statistics derived from the typical costs. Table 4-1 summarizes these statistics.

Table 4-1 Derived Statistics Reported by perfex -y

Statistic Title	Meaning or Use
Graduated instructions per cycle	When the R10000 is used to best advantage, this exceeds 1.0. When it is below 1.0, the CPU is idling some of the time.
Graduated floating point instructions per cycle	Relative density of floating-point operations in the program.
Graduated loads & stores per cycle	Relative density of memory-access in the program.
Graduated loads & stores per floating point instruction	Helps characterize the program as data processing versus mathematical.
Mispredicted branches ÷ Decoded branches	Important measure of the effectiveness of branch prediction, and of code quality.
Graduated loads ÷ Issued loads	When less than 1.0, shows that loads are being reissued because of cache misses.
Graduated stores ÷ Issued stores	When less than 1.0, shows that stores are being reissued because of cache misses or contention between threads or between CPUs.
Data mispredictions ÷ Data scache hits	The count of data misprediction from scache way prediction, as a fraction of all secondary data cache misses.
Instruction mispredictions ÷ Instruction scache hits	The count of instruction misprediction from scache way prediction, as a fraction of all secondary instruction cache misses.
L1 Cache Line Reuse	The average number of times that a primary data cache line is used after it has been moved into the cache. Calculated as graduated loads plus graduated stores minus primary data cache misses, divided by primary data cache misses.
L2 Cache Line Reuse	The average number of times that a secondary data cache line is used after it has been moved into the cache. Calculated as primary data cache misses minus secondary data cache misses, divided by secondary data cache misses.

Table 4-1 (continued) Derived Statistics Reported by perfex -y

Statistic Title	Meaning or Use
L1 Data Cache Hit Rate	The fraction of data accesses satisfied from the L1 data cache. Calculated as 1.0 - (L1 data cache misses ÷ (graduated loads + graduated stores)).
L2 Data Cache Hit Rate	The fraction of data accesses satisfied from the L2 cache. Calculated as 1.0 - (L2 data cache misses ÷ primary data cache misses).
Time accessing memory ÷ Total time	A key measure of time spent idling, waiting for operands. Calculated as the sum of the typical costs of graduated loads and stores, L1 data cache misses, L2 data cache misses, and TLB misses, all divided by the total run time in cycles.
L1-L2 bandwidth used (MBps, average per process)	The amount of data moved between the L1 and L2 data caches, divided by the total run time. The amount of data is taken as: L1 data cache misses times L1 cache line size, plus quadwords written back from L1 data cache times the size of a quadword (16 bytes). For parallel programs, the counts are aggregates over all threads, divided by number of threads. Multiply by the number of threads for total program bandwidth.
Memory bandwidth used (MBps, average per process)	The amount of data moved between L2 cache and main memory, divided by the total run time. The amount of data is taken as: L2 data cache misses times L2 cache line size, plus quadwords written back from L2 cache times the size of a quadword (16 bytes). For parallel programs, the counts are aggregates over all threads, divided by number of threads. Multiply by the number of threads to get the total program bandwidth.
MFLOPS (average per process)	The ratio of graduated floating-point instructions and total run time. Note that a multiply-add carries out two operations, but counts as only one instruction, so this statistic can be an underestimate. For parallel programs, the counts are aggregates over all threads, divided by number of threads. Multiply by the number of threads to get the total program rate.

These statistics give you useful hints about performance problems in your program. For example:

- The cache hit-rate statistics tell you how cache friendly your program is. Because a secondary cache miss is much more expensive than a cache hit, the L2 data cache hit rate needs to be close to 1.0 to indicate that the program is not paying a large penalty for the cache misses. Values of 0.95 and above indicate good cache performance. (For Example 4-3, the rate is 0.53, confirming cache problems in this program.)
- The memory bandwidth used indicates the load that the program places on the SN0 distributed memory architecture. Memory access passes through the hub chip (see "Understanding Scalable Shared Memory" on page 6), which serves two CPUs and which saturates at aggregate rates of 620 MBps.
- Graduated instructions per cycle less than 1.0 indicate the CPU is stalling often for lack of some resource. Look for causes in: the fraction of mispredicted branches, the fraction of time spent accessing memory, and the different cache statistics; and look in the raw data at the proportion of failed store conditionals.

Processing perfex Output

The output of *perfex* goes to the standard output stream, and you can process it further with any of the normal IRIX utilities. A shell script can capture the output of a *perfex* run in a variable and display it, or the output can be filtered using any utility. For an example see Example C-7 on page 298.

Collecting Data over Part of a Run

It is easy to apply *perfex* to an unchanged program, but the data comes from the entire run of the program. Often you want to profile only a particular section of the program—to avoid counting setup work, or to avoid profiling the time spent "warming up" the working sets of cache and virtual memory, or to profile one particular phase of the algorithm.

If you are willing to modify the program, you can use the library interface to *perfex*, documented in the libperfex(3) reference page. You insert one library call to initiate counting and another to terminate it and retrieve the counts. You can perform specific counts of only one or two events; there is no dynamic equivalent to *perfex -a*. The program must then be linked with the *libperfex* library:

```
% f77 -o modules... -lperfex
```

Using perfex with MPI

You can apply *perfex* to a program that uses the MPI message-passing library (MPI is summarized in "Message-Passing Models MPI and PVM" on page 195). The key to applying a tool like *perfex* with MPI is to apply *mpirun* to the tool command. In the following example, *mpirun* is used to start a program under *perfex*:

```
mpirun -np 4 perfex -mp -o afile a.out
```

This example starts copies of *perfex* on four CPUs. Each of them, in turn, invokes the program *a.out*, and writes its analysis report into the file *afile*. It is best to use the *perfex* option *-o* to write output to a file, because *mpirun* redirects the standard output and error streams. For a tip on how to collect standard output from *perfex*, see Example 8-31 on page 254.

Using SpeedShop

The SpeedShop package supports program profiling. You use profiling to find out exactly where a program spends its time, that is, in precisely which procedures or lines of code. Then you can concentrate your efforts on those areas of code where there is the most to be gained.

The SpeedShop package supports three different methods of profiling:

- *Sampling*: The unmodified subject program is rhythmically interrupted by some time base, and the program counter is recorded in a trace file on each interruption.

  Speedshop can use the system timer or any of the R10000 counter events as its time base. Different time bases produce different kinds of information about the program's behavior.

- *Ideal time*: A copy of the program binary is modified to put trap instructions at the end of every basic block. During execution, the exact number of uses of each basic block is counted.

  An ideal time profile is an exact profile of program behavior. A variety of reports can be printed from the trace of an ideal count run.

- *Exception trace*: Not really a profiling method, this records only floating-point exceptions and their locations.

Either method of profiling, sampling or ideal time, can be applied to multiprocessor runs just as easily as it is applied to single-CPU runs. Each thread of an application maintains its own trace information, and the histograms can be printed individually or merged in any combination and printed as one profile.

SpeedShop has three parts:

- *ssrun* performs "experiments" (sampling runs) and collects data.
- The *ssapi* library interface allows you to insert caliper points into a program to profile specific sections of code or phases of execution.
- *prof* processes trace data and prepares reports.

These programs are documented in the reference pages listed under "Related Reference Pages" on page xxxi.

Taking Sampled Profiles

Similar to *perfex*, the *ssrun* command executes the subject program and collects information about the run. However, where *perfex* uses the R10000 event counters to collect data, *ssrun* interrupts the program at regular intervals. This is a statistical sampling method. The time base is the independent variable and the program state is the dependent variable. The output describes the program's behavior as a function of the time base.

Understanding Sample Time Bases

The quality of sampling depends on the time base that sets the sampling interval. The more frequent the interruptions, the better the data collected, and the greater the effect on the run time of the program. The available time bases are listed in Table 4-2.

Table 4-2 SpeedShop Sampling Time Bases

ssrun Option	Time Base	Effect and Use
-usertime	30ms timer	Coarsest resolution; experiment runs quickly and output file is small; some bugs noted in speedshop(1).
-pcsamp[x] -fpcsamp[x]	10 ms timer 1 ms timer	Moderate resolution; functions that cause cache misses or page faults are emphasized. Suffix x for 32-bit counts.
-gi_hwc -fgi_hwc	32771 insts 6553 insts	Fine-grain resolution based on graduated instructions. Emphasizes functions that burn a lot of instructions.
-cy_hwc -fcy_hwc	16411 clocks 3779 clocks	Fine-grain resolution based on elapsed cycles. Emphasizes functions with cache misses and mispredicted branches.
-ic_hwc -fic_hwc	2053 icache miss 419 icache miss	Granularity depends on program behavior. Emphasizes code that doesn't fit in L1 cache.
-isc_hwc -fisc_hwc	131 scache miss 29 scache miss	Granularity depends on program behavior. Emphasizes code that doesn't fit in L2 cache.
-dc_hwc -fdc_hwc	2053 dcache miss 419 dcache miss	Granularity depends on program behavior. Emphasizes code that causes L1 cache data misses.
-dsc_hwc -fdsc_hwc	131 scache miss 29 scache miss	Granularity depends on program behavior. Emphasizes code that causes L2 cache data misses.
-tlb_hwc -ftlb_hwc	257 TLB miss 53 TLB miss	Granularity depends on program behavior. Emphasizes code that causes page faults.
-gfp_hwc -fgfp_hwc	32771 fp insts 6553 fp insts	Granularity depends on program behavior. Emphasizes code that performs heavy FP calculation.
-prof_hwc	user-set	Hardware counter and overflow value from environment variables.

In general, each time base discovers the program PC most often in the code that consumes the most units of that time base, as follows:

- The time bases that reflect actual elapsed time (*-usertime*, *-pcsamp*, *-cy_hwc*) find the PC most often in the code where the program spends the most elapsed time. Time may be spent in that code because the code is executed often; but it might be spent there because those instructions are processed slowly owing to cache misses, contention for memory or locks, or failed branch prediction. Use these time bases to get an overview of the program and to find major trouble spots.
- The time bases that reflect instruction counts (*-gi_hwc*, *-gfp_hwc*) find the PC most often in the code that actually performs the most instructions. Use these to find the code that could benefit most from a more efficient algorithm, without regard to memory access issues.
- The time bases that reflect memory access (*-dc_hwc*, *-sc_hwc*, *-tlb_hwc*) find the PC most often in the code that has to wait for its data to be brought in from another level of the memory hierarchy. Use these to find memory access problems.
- The time bases that reflect code access (*-ic_hwc*, *-isc_hwc*) find the PC most often in the code that has to be fetched from memory when it is called. Use these to pinpoint functions that could be reorganized for better locality, or to see when automatic inlining has gone too far.

Sampling through Hardware Event Counters

Most of the sample time bases listed in Table 4-2 are based on the R10000 event counters (see "Analyzing Performance with Perfex" on page 54 for an overview, and Appendix B, "R10000 Counter Event Types," for details). Only the *-usertime* and -[*f*]*pcsamp* experiments can be run on other types of CPU.

Event counters can be programmed to generate an interrupt after counting a specific number of events. For example, if you choose the *-gi_hwc* sample time base, *ssrun* programs Counter 0 Event 15, graduated instructions, to overflow after 32,771 counts. Each time the counter overflows, the CPU traps to an interrupt routine, which samples the program state, reloads the counter, and restarts the program. The most commonly useful counters can be used directly with the *ssrun* options listed in Table 4-2.

Performing ssrun Experiments

It is easy to perform a sampling experiment. Example 4-4 shows running an experiment on program *adi2.f*.

Example 4-4 Performing an ssrun Experiment

```
% ssrun -fpcsamp adi2
 Time:      7.619 seconds
 Checksum:  5.6160428338E+06
```

The output file of samples is left in a file with a default name compounded from the subject program name, the timebase option, and the process ID to ensure uniqueness. (The exact rules for the output filename are spelled out in reference page speedshop(1).) The output of Example 4-4 might leave a file named *adi2.fpcsamp.m4885*.

You can specify part of the name of the output file. (This is important when, for example, you are automating an experiment with a shell script.) You do this by putting the desired filename and directory in environment variables. The shell script *ssruno*, shown in Example C-6 on page 297, runs an experiment with the output directory and base filename specified. Example 4-5 shows possible output from the use of *ssruno*.

Example 4-5 Example Run of ssruno

```
% ssruno -d /var/tmp -o adi2 -cy_hwc adi2
ssrun -cy_hwc adi2 ...
...... ssrun ends.
-rw-r--r--    1 guest    guest      18480 Dec 17 16:25 /var/tmp/adi2.m5227
```

Sampling Through Other Hardware Counters

In addition to the experiment types listed in Table 4-2, a catchall experiment type, *-prof_hwc*, permits you to design an experiment that uses as its time base, any R10000 counter and any overflow value. You specify the counter number and overflow value using environment variables. (See the speedshop(1) reference page for details.)

As an example, one countable event is an update to a shared cache line (see "Store or Prefetch-Exclusive to Shared Block in Scache (Event 31)" on page 282). A high number of these events in a *perfex* run would lead you to suspect the program is being slowed by memory contention for shared cache lines. But where in the program are the conflicting memory accesses occurring? You can perform an *ssrun* sampling experiment based on *-prof_hwc*, setting the following environment variables:

- _SPEEDSHOP_HWC_COUNTER_NUMBER to 31
- _SPEEDSHOP_HWC_COUNTER_OVERFLOW to a count that will produce at least 100 samples during the run of the program—in other words, the total count that *perfex* reports for event 31, divided by 100 or more.

This experiment will tend to find the program in those statements that update shared cache lines; hence it should highlight the code that suffers from memory contention.

Displaying Profile Reports from Sampling

The output of any SpeedShop experiment is a trace file whose contents are binary samples of program state. You always use the *prof* command to display information about the program run, based on a trace file. Although it can produce a variety of reports, *prof* by default displays a list of routines (functions and procedures), ordered from the routine with the most samples to ones with the fewest. An example of a default report appears in Example 4-6.

Example 4-6 Default prof Report from ssrun Experiment

```
% prof adi2.fpcsamp.4885]
------------------------------------------------------------------------------
Profile listing generated Sat Jan  4 10:28:11 1997
    with:       prof adi2.fpcsamp.4885
------------------------------------------------------------------------------
samples   time    CPU    FPU   Clock   N-cpu  S-interval Countsize
   8574   8.6s R10000 R10010 196.0MHz   1      1.0ms     2(bytes)
Each sample covers 4 bytes for every  1.0ms ( 0.01% of 8.5740s)
------------------------------------------------------------------------------
  -p[rocedures] using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.
------------------------------------------------------------------------------
samples   time(%)      cum time(%)      procedure (dso:file)
   6688   6.7s( 78.0)  6.7s( 78.0)         zsweep (adi2:adi2.f)
    671  0.67s(  7.8)  7.4s( 85.8)         xsweep (adi2:adi2.f)
    662  0.66s(  7.7)    8s( 93.6)         ysweep (adi2:adi2.f)
    208  0.21s(  2.4)  8.2s( 96.0)       fake_adi (adi2:adi2.f)
    178  0.18s(  2.1)  8.4s( 98.1)         irand_ (/usr/lib32/libftn.so:../../libF77/rand_.c)
    166  0.17s(  1.9)  8.6s(100.0)          rand_ (/usr/lib32/libftn.so:../../libF77/rand_.c)
      1 0.001s(  0.0)  8.6s(100.0)      __oserror (/usr/lib32/libc.so.1:oserror.c)
   8574   8.6s(100.0)  8.6s(100.0)          TOTAL
```

Even this simple profile makes it clear that, in this program, you should focus on the routine **zsweep**, because it consumes almost 80% of the run time of the program.

For finer detail, use the *-heavy* (or simply *-h*) option. This supplements the basic report with a list of individual source line numbers, ordered by frequency, as shown in Example 4-7.

Example 4-7 Profile at the Source Line Level Using prof -heavy

```
-------------------------------------------------------------------------------
  -h[eavy] using pc-sampling.
     Sorted in descending order by the number of samples in each line.
     Lines with no samples are excluded.
-------------------------------------------------------------------------------
samples   time(%)      cum time(%)      procedure (file:line)
   3405   3.4s( 39.7)  3.4s( 39.7)         zsweep (adi2.f:122)
   3226   3.2s( 37.6)  6.6s( 77.3)         zsweep (adi2.f:126)
    425  0.42s(  5.0)  7.1s( 82.3)         xsweep (adi2.f:80)
    387  0.39s(  4.5)  7.4s( 86.8)         ysweep (adi2.f:101)
    273  0.27s(  3.2)  7.7s( 90.0)         ysweep (adi2.f:105)
    246  0.25s(  2.9)    8s( 92.9)         xsweep (adi2.f:84)
    167  0.17s(  1.9)  8.1s( 94.8)         irand_ (../../libF77/rand_.c:62)
    163  0.16s(  1.9)  8.3s( 96.7)       fake_adi (adi2.f:18)
    160  0.16s(  1.9)  8.5s( 98.6)          rand_ (../../libF77/rand_.c:69)
     45 0.045s(  0.5)  8.5s( 99.1)       fake_adi (adi2.f:59)
     32 0.032s(  0.4)  8.5s( 99.5)         zsweep (adi2.f:113)
     21 0.021s(  0.2)  8.5s( 99.7)         zsweep (adi2.f:121)
     11 0.011s(  0.1)  8.6s( 99.8)         irand_ (../../libF77/rand_.c:63)
      6 0.006s(  0.1)  8.6s( 99.9)          rand_ (../../libF77/rand_.c:67)
      4 0.004s(  0.0)  8.6s(100.0)         zsweep (adi2.f:125)
      1 0.001s(  0.0)  8.6s(100.0)         ysweep (adi2.f:104)
      1 0.001s(  0.0)  8.6s(100.0)         ysweep (adi2.f:100)
      1 0.001s(  0.0)  8.6s(100.0)      __oserror (oserror.c:127)
   8574   8.6s(100.0)  8.6s(100.0)          TOTAL
```

From this listing it is clear that lines 122 and 126 warrant close inspection. Even finer detail can be obtained with the *-source* option, which lists the source code and disassembled machine code, indicating sample hits on specific instructions.

Using Ideal Time Profiling

The other type of profiling is called ideal time, or basic block, profiling. (A *basic block* is a compiler term for any section of code that has only one entrance and one exit. Any program can be decomposed into basic blocks.) An ideal time profile is not based on statistical sampling. Instead, it is based on an exact count of the number of times each basic block in the program is entered during a run.

Capturing an Ideal Time Trace

To create an ideal profile, *ssrun* copies the executable program and modifies the copy to contain code that records the entry to each basic block. Not only the executable itself but all the dynamic shared objects (standard libraries linked at run time) used by the program are also copied and instrumented. The instrumented executable and libraries are linked statically and run. The output trace file contains precise counts of program execution. Example 4-8 shows an ideal time experiment run.

Example 4-8 Ideal Time Profile Run

```
% ssrun -ideal adi2
Beginning libraries
        /usr/lib32/libssrt.so
        /usr/lib32/libss.so
        /usr/lib32/libfastm.so
        /usr/lib32/libftn.so
        /usr/lib32/libm.so
        /usr/lib32/libc.so.1
Ending libraries, beginning "adi2"
 Time:      8.291 seconds
 Checksum:  5.6160428338E+06
```

The number of times each basic block is encountered is recorded in an experiment file, which is named according to the usual rules of *ssrun* (see "Performing ssrun Experiments" on page 65).

Default Ideal Time Profile

The ideal time trace file is displayed using *prof*, just as for a sampled run. The default report lists every routine the program calls, including library routines. The report is ordered by the count of instructions executed by each procedure, from most to least. An example, edited to shorten the display, is shown in Example 4-9.

Example 4-9 Default Report of Ideal Time Profile

```
%prof adi2.ideal.m5064
------------------------------------------------------------------------
SpeedShop profile listing generated Thu Jun 25 11:42:33 1998
   prof adi2.ideal.m5064
                    adi2 (n32): Target program
                         ideal: Experiment name
                         it:cu: Marching orders
```

```
              R10000 / R10010: CPU / FPU
                          16: Number of CPUs
                         250: Clock frequency (MHz.)
  Experiment notes--
    From file adi2.ideal.m5064:
    Caliper point 0 at target begin, PID 5064
    /usr/people/cortesi/ADI/adi2.pixie
    Caliper point 1 at exit(0)
------------------------------------------------------------------------
Summary of ideal time data (ideal)--
                    1091506962: Total number of instructions executed
                    1218887512: Total computed cycles
                         4.876: Total computed execution time (secs.)
                         1.117: Average cycles / instruction
------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
------------------------------------------------------------------------
[index] excl.secs excl.%  cum.%      cycles instructions calls  function  (dso: file, line)

 [1]       1.386  28.4%   28.4%   346619904   304807936   32768  xsweep (adi2: adi2.f, 71)
 [2]       1.386  28.4%   56.9%   346619904   304807936   32768  ysweep (adi2: adi2.f, 92)
 [3]       1.386  28.4%   85.3%   346619904   304807936   32768  zsweep (adi2: adi2.f, 113)
 [4]       0.540  11.1%   96.4%   135022759   111785107       1  fake_adi (adi2: adi2.f, 1)
 [5]       0.101   2.1%   98.5%    25165824    35651584 2097152  rand_ (libftn.so...)
 [6]       0.067   1.4%   99.8%    16777216    27262976 2097152  irand_ (libftn.so...)
 [7]       0.002   0.0%   99.9%      505881      589657    1827  general_find_symbol (...)
 [8]       0.001   0.0%   99.9%      364927      378813    3512  resolve_relocations (...)
 [9]       0.001   0.0%   99.9%      214140      185056    1836  elfhash (rld: obj.c, 1211)
[10]       0.001   0.0%   99.9%      164195      182151    1846  find_symbol_in_object (...)
[11]       0.001   0.0%   99.9%      132844      177140    6328  obj_dynsym_got (...)
[12]       0.000   0.0%  100.0%       95570      117237    1804  resolving (rld: rld.c, 1893)
[13]       0.000   0.0%  100.0%       90941      112852    1825  resolve_symbol (rld:...)
[14]       0.000   0.0%  100.0%       86891      211175    3990  strcmp (rld: strcmp.s, 34)
[15]       0.000   0.0%  100.0%       75092       77499       1  fix_all_defineds (...)
[16]       0.000   0.0%  100.0%       67523       57684    9054  next_obj (rld: rld.c, 2712)
[17]       0.000   0.0%  100.0%       65813       59722       6  search_for_externals (...)
[18]       0.000   0.0%  100.0%       53431       73229    1780  find_first_object_to_search
[19]       0.000   0.0%  100.0%       42153       35668    3243  obj_set_dynsym_got (...)
[20] ...
```

Interpreting the Ideal Time Report

The key information items in a report like the one in Example 4-9 are:

- The exact count of machine instructions executed by the code of each function or procedure
- The percentage of the total execution cycles consumed in each function or procedure

An ideal profile shows precisely which statements are most often executed, giving you an exact view of the algorithmic complexity of the program. However, an ideal profile is based on the ideal, or standard, time that each instruction ought to take. It does not necessarily reflect where a program spends elapsed time. The profile cannot take into account the ability of the CPU to overlap instructions, which can shorten the elapsed time. More important, it cannot take into account instruction delays caused by cache and TLB misses, which can greatly lengthen elapsed time.

The assumption of ideal time explains why the results of the profile in Example 4-9 are so startlingly different from that of the sampling profile of the same program in Example 4-6 on page 67. From Example 4-9 you would expect **zsweep** to take exactly the same amount of run time as **ysweep** or **xsweep**—the instruction counts, and hence ideal times, are identical. Yet when sampling is used, **zsweep** takes ten times as much time as the other procedures.

The only explanation for such a difference is that some of the instructions in **zsweep** take longer than the ideal time to execute, so that the sampling run is more likely to find the PC in **zsweep**. The inference is that the code of **zsweep** encounters many more cache misses, or possibly TLB misses, than the rest of the program. (On machines without the R10000 CPU's hardware profiling registers, such a comparison is the only profiling method that can identify cache problems.)

Removing Clutter from the Report

It is quickly apparent that only the first few lines of the ideal time report are useful for tuning. The dozens of library functions the program called are of little interest. You can eliminate the clutter from the report by applying the *-quit* option to cut the report off after showing functions that used significant time, as shown in edited form in Example 4-10.

Example 4-10 Ideal Time Report Truncated with -quit

```
%prof -q 2% adi2.ideal.m5064
------------------------------------------------------------------------
SpeedShop profile listing generated Thu Jun 25 12:58:57 1998
   prof -q 2% adi2.ideal.m5064
...
Summary of ideal time data (ideal)--
                    1091506962: Total number of instructions executed
                    1218887512: Total computed cycles
                         4.876: Total computed execution time (secs.)
                         1.117: Average cycles / instruction
------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
------------------------------------------------------------------------
[index] excl.secs excl.%  cum.%  cycles    instructions  calls function (dso: file, line)

[1]     1.386     28.4%   28.4%  346619904  304807936    32768  xsweep (adi2: adi2.f, 71)
[2]     1.386     28.4%   56.9%  346619904  304807936    32768  ysweep (adi2: adi2.f, 92)
[3]     1.386     28.4%   85.3%  346619904  304807936    32768  zsweep (adi2: adi2.f, 113)
[4]     0.540     11.1%   96.4%  135022759  111785107        1  fake_adi (adi2: adi2.f, 1)
[5]     0.101      2.1%   98.5%   25165824   35651584  2097152  rand_ ( )
```

Including Line-Level Detail

The *-heavy* option appends a list of source lines, sorted by their consumption of ideal instruction cycles. The combination of *-q* and *-h* is shown in Example 4-11.

Example 4-11 Ideal Time Report by Lines

```
------------------------------------------------------------------------
SpeedShop profile listing generated Thu Jun 25 13:03:06 1998
   prof -h -q 2% adi2.ideal.m5064
...
------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
------------------------------------------------------------------------
...function list same as Example 4-10...
------------------------------------------------------------------------
Line list, in descending order by time
------------------------------------------------------------------------
  excl.secs      %    cum.%        cycles   invocations  function (dso: file, line)
      0.565   11.6%   11.6%     141273180       4161536  xsweep (adi2: adi2.f, 80)
      0.565   11.6%   23.2%     141273180       4161536  ysweep (adi2: adi2.f, 101)
      0.565   11.6%   34.8%     141273180       4161536  zsweep (adi2: adi2.f, 122)
      0.552   11.3%   46.1%     137925189       4161536  xsweep (adi2: adi2.f, 84)
      0.552   11.3%   57.4%     137925189       4161536  ysweep (adi2: adi2.f, 105)
      0.552   11.3%   68.7%     137925189       4161536  zsweep (adi2: adi2.f, 126)
      0.196    4.0%   72.7%      49041090       2097152  fake_adi (adi2: adi2.f, 59)
      0.194    4.0%   76.7%      48416848       2097152  fake_adi (adi2: adi2.f, 18)
      0.152    3.1%   79.8%      37948124       4161536  xsweep (adi2: adi2.f, 79)
      0.152    3.1%   82.9%      37948124       4161536  ysweep (adi2: adi2.f, 100)
      0.152    3.1%   86.1%      37948124       4161536  zsweep (adi2: adi2.f, 121)
      0.116    2.4%   88.4%      28896699       4161536  ysweep (adi2: adi2.f, 104)
      0.116    2.4%   90.8%      28896699       4161536  zsweep (adi2: adi2.f, 125)
      0.116    2.4%   93.2%      28896699       4161536  xsweep (adi2: adi2.f, 83)
```

You can use the *-lines* option instead of *-heavy*. This lists the source lines in their source sequence, grouped by procedure, with the procedures ordered by decreasing time. This option helps you see the expensive statements in their context. However, the option does not combine well with the *-q* option. In order to use *-lines* yet avoid voluminous output, choose specific procedures that you want to analyze and restrict the display to only those procedures using *-only*, as shown in Example 4-12.

Example 4-12 Ideal Time Profile Using -lines and -only Options

```
------------------------------------------------------------------------
SpeedShop profile listing generated Thu Jun 25 13:16:54 1998
   prof -l -only zsweep -o xsweep adi2.ideal.m5064
...
------------------------------------------------------------------------
Function list, in descending order by exclusive ideal time
------------------------------------------------------------------------
[index] excl.secs excl.% cum.%     cycles  instructions calls function (dso: file, line)
 [1]    1.386     28.4%  28.4%  346619904  304807936    32768  xsweep (adi2: adi2.f, 71)
 [2]    1.386     28.4%  56.9%  346619904  304807936    32768  zsweep (adi2: adi2.f, 113)
------------------------------------------------------------------------
Line list, in descending order by function-time and then line number
------------------------------------------------------------------------
  excl.secs  excl.%   cum.%        cycles   invocations  function (dso: file, line)
      0.002    0.0%    0.0%        412872         32768  xsweep (adi2: adi2.f, 71)
      0.152    3.1%    3.1%      37948124       4161536  xsweep (adi2: adi2.f, 79)
      0.565   11.6%   14.7%     141273180       4161536  xsweep (adi2: adi2.f, 80)
      0.116    2.4%   17.1%      28896699       4161536  xsweep (adi2: adi2.f, 83)
      0.552   11.3%   28.4%     137925189       4161536  xsweep (adi2: adi2.f, 84)
      0.001    0.0%   28.4%        163840         32768  xsweep (adi2: adi2.f, 87)
      0.002    0.0%   28.5%        412872         32768  zsweep (adi2: adi2.f, 113)
      0.152    3.1%   31.6%      37948124       4161536  zsweep (adi2: adi2.f, 121)
      0.565   11.6%   43.2%     141273180       4161536  zsweep (adi2: adi2.f, 122)
      0.116    2.4%   45.5%      28896699       4161536  zsweep (adi2: adi2.f, 125)
      0.552   11.3%   56.9%     137925189       4161536  zsweep (adi2: adi2.f, 126)
      0.001    0.0%   56.9%        163840         32768  zsweep (adi2: adi2.f, 129)
```

Any simple relationship between source statement numbers and the executable code is destroyed during optimization. The statement numbers listed in a profile correspond very nicely to the source code when the program was compiled without optimization (*-O0* compiler option). When the program was compiled with optimization, the source statement numbers in the profile are approximate, and you may have difficulty relating the profile to the code.

Creating a Compiler Feedback File

The information in an ideal-time profile includes exact counts of how often every conditional branch was taken or not taken. The compiler can use this information to generate optimal branch code. You use *prof* to extract the branch statistics into a compiler feedback file using the *-feedback* option:

```
prof -feedback program.ideal.n
```

The usual display is produced on standard output, but prof also writes a *program.cfb* file. You provide this as input to the compiler using the *-fb* compiler option, as described in "Passing a Feedback File" on page 124.

Displaying Operation Counts

Because ideal profiling counts the instructions executed by the program, it can provide all sorts of interesting information about the program. Use the *-archinfo* option to get a detailed census of the most arcane details of the program's execution. An edited version is shown in Example 4-13.

Example 4-13 Ideal Time Architecture Information Report

```
------------------------------------------------------------------------
SpeedShop profile listing generated Thu Jun 25 13:34:40 1998
   prof -archinfo adi2.ideal.m5064
...
Integer register usage
----------------------
 register      use count      %    base count      %   dest. count      %
      r00      342234300  17.76%            0   0.00%      4334788   0.22%
      r01       10878870   0.56%     29178987   1.51%      2223380   0.12%
      r02       59266446   3.08%        41971   0.00%      2308302   0.12%
      r03 ...
Floating-point register usage
-----------------------------
 register      use count      %   dest. count      %
      f00       62521432  27.29%      6291491   2.75%
      f01       37552180  16.39%     14581780   6.36%
      f02       29163562  12.73%           21   0.00%
      f03 ...
Instruction execution statistics
--------------------------------
         1091506962: instructions executed
           56230002: floating point operations (11.5331 Mflops @ 250 MHz)
          400759124: integer operations (82.1977 M intops @ 250 MHz)
```

```
...
Instruction counts
------------------
    Name   executions   exc. %  cum. %   instructions  inst. %   cum.%

      lw    303415993   28.69%  28.69%           1415   12.93%  12.93%
   addiu    115696249   10.94%  39.64%            992    9.07%  22.00%
    addu     96419728    9.12%  48.75%            352    3.22%  25.22%
...
  madd.d     12484612    1.18%  93.82%              5    0.05%  50.54%
...
```

Among the points to be learned with *-archinfo* are these:

- The count and average rate of floating-point operations. Note that these counts differ from the ones reported by *perfex*. R10000 Event 21 counts floating-point *instructions*, not floating point *operations*. The multiply-add instruction carries out two floating-point operations; *perfex* counts it as 1 and *prof* as 2.
- The count and average rate of integer operations, allowing you to judge whether these (such as index calculations) are important to performance.
- Exact counts of specific opcodes executed, including a tally of those troublesome multiply-add (madd.d) instructions.

Profiling the Call Hierarchy

The profiles discussed so far do not reflect the call hierarchy of the program. If the routine **zsweep** were called from two different locations, you could not tell how much time resulted from the calls at each location; you only know the total time spent in **zsweep**. If you could learn that, say, most of the calls came from the first location, it would affect how you tuned the program. For example, you might try inlining the call to the first location, but not the second. Or, if you want to parallelize the program, knowing that the first location is where most of the time is spent, you might consider parallelizing the calls to **zsweep**, rather than trying to parallelize the **zsweep** routine itself.

Only two types of SpeedShop profiles capture sufficient information to reveal the call hierarchy: ideal counting, and one of the sampling time bases.

Displaying Ideal Time Call Hierarchy

You can request call hierarchy information in an ideal time report using the *-butterfly* flag (and the *-q* flag to reduce clutter). This produces the usual report (like Example 4-10), with an additional section which, in the prof(1) reference page, is called the "butterfly report." An edited extract of a butterfly report is shown in Example 4-14.

Example 4-14 Extract from a Butterfly Report

```
-----------------------------------------------------------
     99.8%   4.867(0000001)                          4.867  __start [1]
[2]  99.8%   4.867     0.0%    0.000         main [2]
                      99.8%    4.867(0000001)         4.867  fake_adi [3]
                       0.0%    0.000(0000005)         0.000  signal [125]
-----------------------------------------------------------
     99.8%   4.867(0000001)                          4.867  main [2]
[3]  99.8%   4.867    11.1%    0.540         fake_adi [3]
                      28.4%    1.386(0032768)         1.386  zsweep [4]
                      28.4%    1.386(0032768)         1.386  ysweep [6]
                      28.4%    1.386(0032768)         1.386  xsweep [5]
                       3.4%    0.168(2097152)         0.168  rand_ [7]
                       0.0%    0.000(0000002)         0.000  e_wsfe [63]
                       0.0%    0.000(0000002)         0.000  s_wsfe64 [68]
                       0.0%    0.000(0000001)         0.000  do_fioxr8v [84]
                       0.0%    0.000(0000001)         0.000  do_fioxr4v [83]
                       0.0%    0.000(0000001)         0.000  s_stop [86]
                       0.0%    0.000(0000002)         0.000  dtime_ [183]
-----------------------------------------------------------
     28.4%   1.386(0032768)                          4.867  fake_adi [3]
[4]  28.4%   1.386    28.4%    1.386         zsweep [4]
-----------------------------------------------------------
     28.4%   1.386(0032768)                          4.867  fake_adi [3]
[5]  28.4%   1.386    28.4%    1.386         xsweep [5]
-----------------------------------------------------------
     28.4%   1.386(0032768)                          4.867  fake_adi [3]
[6]  28.4%   1.386    28.4%    1.386         ysweep [6]
-----------------------------------------------------------
      3.4%   0.168(2097152)                          4.867  fake_adi [3]
[7]   3.4%   0.168     2.1%    0.101         rand_ [7]
                       1.4%    0.067(2097152)         0.067  irand_ [8]
```

There is a block of information for each routine in the program. A number, shown in brackets (for example, `[3]`), is assigned to each routine for reference. Examine the second, and largest, section in Example 4-14, the one for routine **fake_adi**.

The line that begins and ends with the reference number (in this example, `[3]`) describes the routine itself. It resembles the following:

```
[3]  99.8%   4.867    11.1%    0.540          fake_adi [3]
```

From left to right it shows:

- The total run time attributed to this routine and its descendants, as a percentage of program run time and as a count (99.8% and 4.867 seconds).
- The amount of run time that was consumed in the code of the routine itself. In this example, the routine code accounted for 11.1% of the run time, or 0.54 seconds. The remaining 88.7% of the time was spent in subroutines that it called.
- The name of the routine in question is **fake_adi**.

The lines that precede this line describe the places that called **fake_adi**. In the example there was only one such place, so only one line appears, resembling the following:

```
99.8%   4.867(0000001)                              4.867  main [2]
```

From left to right, it shows:

- All calls made to **fake_adi** from this location consumed 99.8%, or 4.867 seconds, of run time.
- How many calls were made from this location (just 1, in this case).
- How much time was consumed by this calling routine.
- The name of the calling routine, in this case, **main**.

When a routine is called from more than one place, these lines show you which locations made the most calls, and how much time the calls from each location cost. This lets you zero in on the important callers and neglect the minor ones.

The lines that follow the central line show the calls this routine makes, and how its time is distributed among them. In Example 4-14, routine **fake_adi** calls 10 other routines, but the bulk of the ideal time is divided among the first three. These lines let you home in on precisely those subroutines that account for the most time, and ignore the rest.

The butterfly report tells you not just which subroutines are consuming the most time, but which execution paths in the program are consuming time. This lets you decide whether you should try to optimize a subroutine, inline the subroutine, or try to eliminate some calls to the subroutine, or redesign the algorithm. For example, suppose you find that the bulk of calls to a general-purpose subroutine come from one place, and on examining that call, you find that all the calls are for one simple case. You could either replace the call with in-line code, or you could add a special-purpose subroutine to handle that case very quickly.

The limitation of this report is that it is based on ideal time, and so does not reflect costs due to cache misses and other elapsed-time delays.

Displaying Usertime Call Hierarchy

To get a hierarchical profile that accounts for elapsed time, use usertime sampling instead of ideal time. As shown in Table 4-2 on page 64, a *-usertime* experiment samples the state of the program every 30 ms. Each sample records the location of the program counter and the entire call stack, showing which routines have been called to reach this point in the program. From this trace data a hierarchical profile can be constructed. (Other sampling experiments record only the PC, not the entire call stack.)

You take a *-usertime* sample exactly like any other ssrun experiment (see "Performing ssrun Experiments" on page 65). The result for sample program *adi2* (edited for brevity) is shown in Example 4-15.

Example 4-15 Usertime Call Hierarchy

```
% ssrun -usertime adi2
 Time:      8.424 seconds
 Checksum:  5.6160428338E+06
% prof -butterfly adi2.usertime.m6836
------------------------------------------------------------------------
SpeedShop profile listing generated Thu Jun 25 16:28:27 1998
   prof -butterfly adi2.usertime.m6836
...
------------------------------------------------------------------------
Summary of statistical callstack sampling data (usertime)--
                          305: Total Samples
                            0: Samples with incomplete traceback
                        9.150: Accumulated Time (secs.)
                         30.0: Sample interval (msecs.)
```

```
-------------------------------------------------------------------------
Function list, in descending order by exclusive time
-------------------------------------------------------------------------
[index]  excl.secs excl.% cum.%   incl.secs incl.%    samples  procedure  (dso: file, line)
[4]      5.670  62.0%   62.0%      5.670      62.0%      189  zsweep (adi2: adi2.f, 113)
[5]      1.410  15.4%   77.4%      1.410      15.4%       47  xsweep (adi2: adi2.f, 71)
[6]      1.380  15.1%   92.5%      1.380      15.1%       46  ysweep (adi2: adi2.f, 92)
[1]      0.510   5.6%   98.0%      9.150     100.0%      305  fake_adi (adi2: adi2.f, 1)
[8]      0.120   1.3%   99.3%      0.120       1.3%        4  irand_ (libftn.so: rand_.c, 62)
[7]      0.060   0.7%  100.0%      0.180       2.0%        6  rand_ (libftn.so: rand_.c, 67)
[2]      0.000   0.0%  100.0%      9.150     100.0%      305  __start (adi2: crt1text.s, 103)
[3]      0.000   0.0%  100.0%      9.150     100.0%      305  main (libftn.so: main.c, 76)
-------------------------------------------------------------------------
Butterfly function list, in descending order by inclusive time
...
-------------------------------------------------------
    100.0%   9.150                    9.150  main
[1] 100.0%   9.150     5.6%  0.510     fake_adi [1]
                     62.0%  5.670    5.670  zsweep
                     15.4%  1.410    1.410  xsweep
                     15.1%  1.380    1.380  ysweep
                      2.0%  0.180    0.180  rand_
-------------------------------------------------------
     62.0%   5.670                    9.150  fake_adi
[4]  62.0%   5.670    62.0%  5.670     zsweep [4]
-------------------------------------------------------
     15.4%   1.410                    9.150  fake_adi
[5]  15.4%   1.410    15.4%  1.410     xsweep [5]
-------------------------------------------------------
     15.1%   1.380                    9.150  fake_adi
[6]  15.1%   1.380    15.1%  1.380     ysweep [6]
-------------------------------------------------------
      2.0%   0.180                    9.150  fake_adi
[7]   2.0%   0.180     0.7%  0.060     rand_ [7]
                      1.3%  0.120    0.120  irand_
-------------------------------------------------------
      1.3%   0.120                    0.180  rand_
[8]   1.3%   0.120     1.3%  0.120     irand_ [8]
```

The information is less detailed than an ideal time profile (only 305 samples over this entire run), but it reflects elapsed time. Compare the butterfly report section for routine **fake_adi** in Example 4-15 to the same section in Example 4-14. When actual time differs so much from ideal time, something external to the algorithm is delaying execution. We will investigate this problem later (see "Identifying Cache Problems with Perfex and SpeedShop" on page 142.)

Using Exception Profiling

A program can handle floating-point exceptions using the **handle_sigfpes** library procedure (documented in the sigfpe(3) reference pages). Many programs do not handle exceptions. If exceptions occur, they still cause hardware traps that are ignored in software. A high number of ignored exceptions can cause a program to run slowly for no apparent reason.

Profiling Exception Frequency

The *ssrun* experiment type *-fpe* creates a trace file that records all floating-point exceptions. Use *prof* to display the data as usual. The report shows the procedures and lines that caused exceptions. In effect, this is a sampling profile in which the sampling time base is the occurrence of an exception.

Use exception profiling to verify that a program does not have significant exceptions. When you know that, you can safely set a higher level of exception handling via the compiler flag *-TENV:X* (see "Permitting Speculative Execution" on page 121).

Understanding Treatment of Underflow Exceptions

When a program does generate a significant number of undetected exceptions, the exceptions are likely to be floating-point underflows (a program is not likely to get through functional testing if it suffers a large number of divide-by-zero, overflow, or invalid operation exceptions). A large number of underflows causes performance problems because each exception generates a trap to the kernel to finish the calculation by setting the result to zero. Although the results are correct, excess system time elapses. On R8000 based systems, the default is to flush underflows to zero in the hardware, avoiding the trap. However, R10000, R5000, and R4400 systems default to trapping on underflow.

If underflow detection is not required for numerical accuracy, underflows can be flushed to zero in hardware. This can be done by methods described in detail in reference page sigfpe(3C). The simplest method is to link in the floating point exception library, *libfpe*, and at run tim to set the TRAP_FPE environment variable to the string UNDERFL=FLUSH_ZERO.

The hardware bit that controls how underflows are handled is located in a part of the R10000 floating-point unit called the floating-point status register (FSR). In addition to setting the way underflows are treated, this register also selects which IEEE rounding mode is used and which IEEE exceptions are enabled (that is, not ignored). Furthermore, it contains the floating-point condition bits, which are set by floating-point compare instructions, and the cause bits, which are set when any floating-point exception occurs. The details of the FSR are documented in the *MIPS R10000 Microprocessor User Guide* listed in "Related Manuals" on page xxix. You can gain programmed access to the FSR using functions described in reference page sigfpe(3C). However, to do so makes your program hardware-dependent.

Using Address Space Profiling

Speedshop and *perfex* profile the execution path, but there is a second dimension to program behavior: data references.

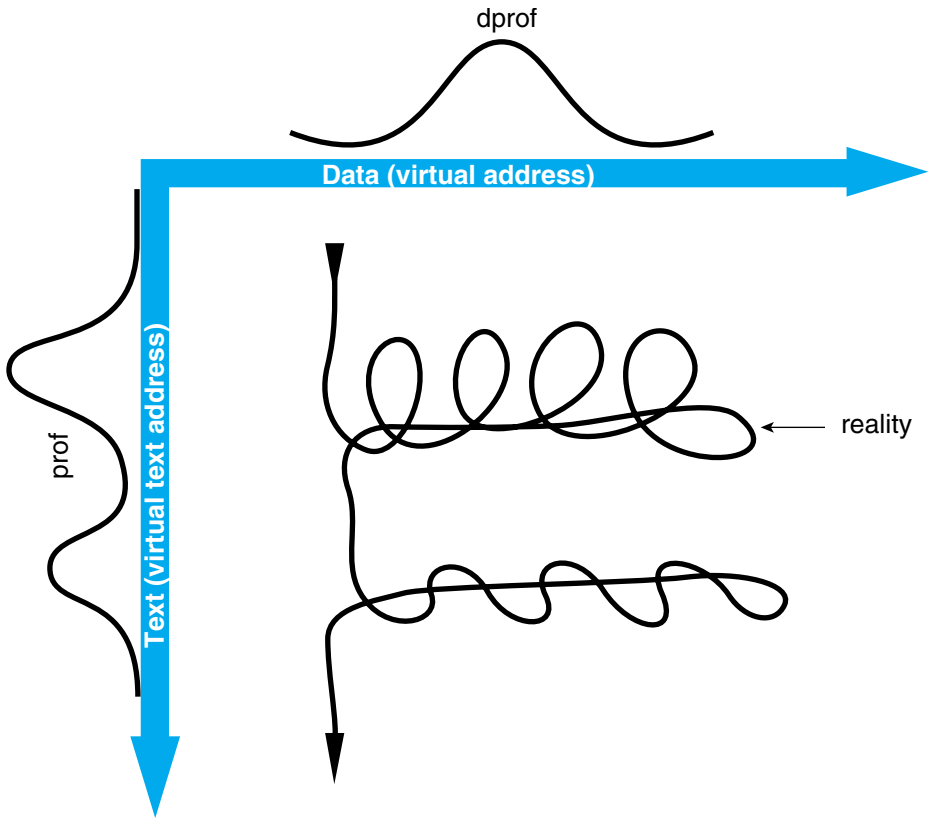


Figure 4-1 Code Residence versus Data reference

Both dimensions are sketched in Figure 4-1. The program executes up and down through the program text. An execution profile depicts the average intensity of the program's residence at any point, but execution moves constantly. At the same time, the program constantly refers to locations in its data.

You can use *dprof* to profile the intensity with which the program uses data memory locations, showing the alternate dimension to that shown by *prof*. You can use *dlook* to find out where in the SN0 architecture the operating system placed a program's pages.

Applying dprof

The *dprof* tool, like *ssrun*, executes a program and samples the program while it runs. Unlike *ssrun*, *dprof* does not sample the PC and stack. It samples the current operand address of the interrupted instruction. It accumulates a histogram of access to data addresses, grouped in units of virtual page. Example 4-16 shows an application of *dprof* to the test program.

Example 4-16 Application of dprof

```
% dprof -hwpc -out adi2.dprof adi2
 Time:     27.289 seconds
 Checksum:  5.6160428338E+06
% ls -l adi2.dprof
-rw-r--r--    1 guest      40118 Dec 18 18:54 adi2.dprof
% cat adi2.dprof
------------------------------------------------
   address      thread       reads          writes
------------------------------------------------
0x0010012000      0           1482              0
0x007eff4000      0          59075           1462
0x007eff8000      0             57             22
0x007effc000      0             69             15
0x007f000000      0             75             18
0x007f004000      0             58             10
0x007f008000      0             65             13
0x007f00c000      0             64             20
0x007f010000      0             76             22
...
0x007ffe0000      0             59             16
0x007ffe4000      0             70              9
0x007ffe8000      0             57             11
0x007ffec000      0             53              8
0x007fff0000      0             56             14
0x007fff4000      0             36              1
```

Each line contains a count of all references to one virtual memory page from one IRIX process (this program has only one process). Note that the addresses at the left increment by 0x04000, or 16 KB, the default size of a virtual page in larger systems under IRIX 6.5.

Interpreting dprof Output

The *dprof* report is based on statistical sampling; it does not record all references. The time base is either the interval timer (option *-itimer*) or the R10000 event cycle counter (option *-hwpc*, available only to an R10000 CPU). Other time base options are supported; see the dprof(1) reference page.

At the default interrupt frequency using the cycle counter, samples are taken of only 1 instruction in 10,000. This produces a coarse sample that is not likely to be repeatable from run to run. However, even this sampling rate slows program execution by almost a factor of three. You can obtain a more detailed sample by specifying a shorter overflow count (option *-ovfl*), but this will extend program execution time proportionately.

The coarse histogram is useful for showing which pages are used. For example, you can plot the data as a histogram. Using *gnuplot* (which is available on the Silicon Graphics Freeware CDROM), a simple plot of total access density, as shown in Figure 4-2, is obtained as follows:

```
% /usr/freeware/bin/gnuplot
                            G N U P L O T
        unix version 3.5
        patchlevel 3.50.1.17, 27 Aug 93
        last modified Fri Aug 27 05:21:33 GMT 1993
        Copyright(C) 1986 - 1993   Thomas Williams, Colin Kelley
        Send comments and requests for help to info-gnuplot@dartmouth.edu
        Send bugs, suggestions and mods to bug-gnuplot@dartmouth.edu
Terminal type set to 'x11'
gnuplot> set logscale y
gnuplot> plot "<tail +4 adi2.dprof|awk '{print NR,$3+$4,$2}'" with box
```

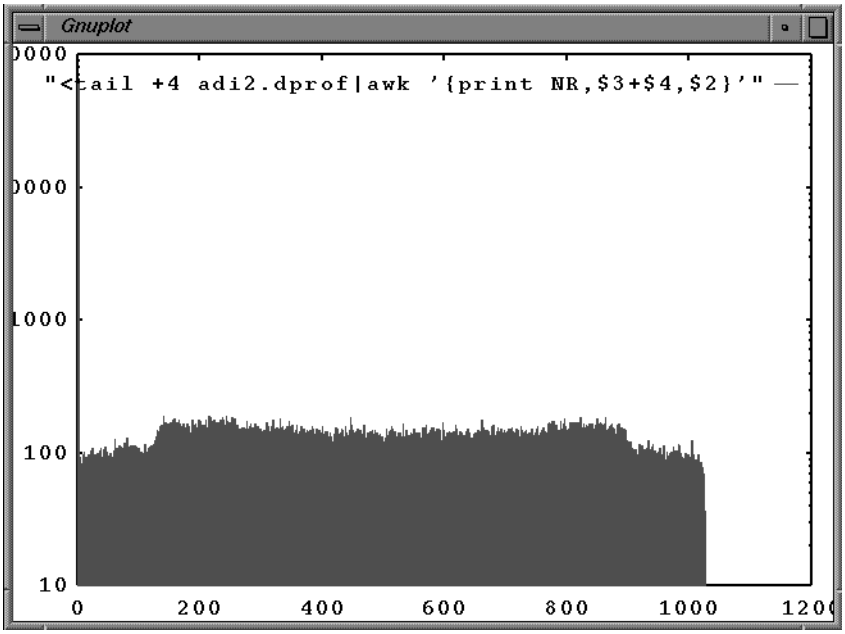


Figure 4-2 On-screen plot of dprof output

The information provided by *dprof* is not of great benefit to a single-threaded application. But for multithreaded applications, it can reveal a lot. Because access to the local memory is faster than access to remote memories on an SN0 system, the best performance on parallel programs that sustain a lot of cache misses is achieved if each thread primarily accesses its local memory. *dprof* gives you the ability to analyze which regions of virtual memory each thread of a program accesses.

The output of *dprof* can also be used as input to *dplace*; see "Assigning Memory Ranges" on page 252.

Applying dlook

The *dlook* program (see the dlook(1) reference page), like *perfex* and *dprof*, takes a second program's command line as its argument. It executes that program, and monitors where the pages of text and data are located. At the end of the run, *dlook* writes a report on the

memory placement to the standard error file, or to a file you specify. Example 4-17 shows the result of applying *dlook* with default options.

Example 4-17 Example of Default dlook Output

```
[farewell 4] dlook adi2
 Time:      8.498 seconds
 Checksum:  5.6160428338E+06
_____________________________________________________________________
Exit  : "adi2" is process 1773 and at ~9.38 second is running on:
        cpu 10 or /hw/module/6/slot/n3/node/cpu/a .
data/heap:
[0x00010010000,0x00010024000]     5 16K pages on /hw/module/6/slot/n3/node
stack:
[0x0007eff0000,0x0007fd80000]   868 16K pages on /hw/module/6/slot/n3/node
[0x0007fd80000,0x0007fff0000]   156 16K pages on /hw/module/6/slot/n1/node
[0x0007fff0000,0x0007fff8000]     2 16K pages on /hw/module/5/slot/n1/node
_____________________________________________________________________
```

The *-mapped* option produces a lengthy list of all memory-mapped regions (such regions are normally mapped by library code, not the program itself). The *-sample* option produces a periodic report during the run, from which you could follow page migration.

Sometimes *dlook* reports a page as being located on device */hw/mem*, instead of a particular module. This means that the page has been added to the program's virtual address space, but because the program has not yet touched the page, it has not been allocated to real memory.

Summary

You can record the dynamic behavior of a program in a number of different ways, simply, quickly, and without modifying the program. You can display the output of any experiment in different ways, revealing the execution "hot spots" in the code and the effect of its subroutine call structure. All the information you need in order to tune the algorithm is readily available.

Chapter 5

Using Basic Compiler Optimizations

Profiling reveals which code to tune. The most important tool for tuning is the compiler. Ideally, the compiler should convert your program to the fastest possible code automatically. But this isn't possible because the compiler does not have enough information to make optimal decisions.

As an example, if one loop iterates many times, it will run fastest if the compiler unrolls the loop and performs software pipelining on the unrolled code. However, if the loop iterates only a few times, the time spent in the elaborate housekeeping code generated by pipelining is never recovered, and the loop actually runs more slowly. With some Fortran DO loops and most C loops, the compiler cannot tell how many times a loop will iterate. What should it do? The answer, of course, is that you have to direct the compiler to apply the right optimizations. To do that, you need to understand the types of optimizations the compiler can make, and the command-line options ("compiler flags") that enable or disable them. These are described in this chapter, in the following sections:

- "Understanding Compiler Options" on page 90 lists the compiler options, or flags, that should be used initially, and discusses the use of a Makefile.
- "Exploiting Software Pipelining" on page 99 describes the uses and control of this important optimization.
- "Informing the Compiler" on page 109 describes the use of compiler flags, pragmas, and coding techniques to give the compiler more information so that it can better optimize and pipeline loops.
- "Exploiting Interprocedural Analysis" on page 125 discusses the concepts and use of IPA and inlining.

The discussion of single-process tuning continues in Chapter 6, "Optimizing Cache Utilization," covering the closely-related topics of loop nest optimization and memory access.

Understanding Compiler Options

The Silicon Graphics compilers are flexible and offer a wide variety of compiler options to control their operation. The options are documented in a multiple set of reference pages, as well as in the compiler manuals listed in "Compiler Manuals" on page xxix.

Recommended Starting Options

Because the optimizations the compilers can perform interact with each other, and because no two programs are alike, it is impossible to specify a fixed set of options that give the best results in all cases. Nevertheless, the following is a good set of compiler flags to start with:

```
-n32 -mips4 -O2 -OPT:IEEE_arithmetic=3 -lfastm -lm
```

The meaning of these options is as follows:

-n32	Use the new 32-bit ABI. Use -64 only when the program needs more than 2 GB of virtual address space
-mips4	Compile code for the R10000 CPU.
-O2	Request the best set of conservative optimizations; that is, those that do not reorder statements and expressions.
-OPT:IEEE_arithmetic=3	Permit optimizations that favor speed over preserving precise numerical rounding rules. If the program proves numerically unstable or inaccurate use *-OPT:IEEE_arithmetic=2* instead.
-lfastm -lm	Link math routines from the optimized math library; take any remaining math routines from the standard library.

The *-n32* and *-mips4* options are discussed in "Selecting an ABI and ISA" on page 46; the library choice under "libfastm Library" on page 50. The others are discussed in the topics in this chapter.

Using this set of flags for all modules of a large program suite may produce acceptable performance. If so, tuning is complete. Otherwise, the performance-critical modules will require compilation with higher levels of optimization than *-O2*. These levels are discussed under "Setting Optimization Level with -On" on page 93.

Compiler Option Groups

The compilers have many options that are specified in the conventional UNIX fashion, as a hyphen followed by a word. For example, optimization level *-O2*; ABI level *-n32*; separate compilation of a module *-c*; request assembler source file *-S*. These options are all documented in the compiler reference pages cc(1), f77(1), f90(1). (You will find that it is useful to keep a printed copy of your compiler's reference page near your keyboard.)

The compilers categorizes other options into groups. These groups of options are documented in separate reference pages. The option groups are listed in Table 5-1.

Table 5-1 Compiler Option Groups

Group Heading	Reference Page	Use
-CLIST, -FLIST	cc(1), f77(1)	Listing controls.
-DEBUG	debug_group(5)	Options for debugging and error-trapping.
-INLINE	ipa(5)	Options related to the standalone inliner.
-IPA	ipa(5)	Options related to interprocedural analysis.
-LANG	cc(1), f77(1)	Language compatibility selections.
-LNO	lno(5)	Options related to the loop-nest optimizer.
-OPT	opt(5)	Specific controls of the global optimizer.
-TARG, -TENV	cc(1), f77(1)	Specify target execution environment features.

The syntax for an option within a group is:

`-`*group*`:`*option*`[=`*value*`][:`*option*`[=`*value*`]]...`

You can specify a group heading more than once; the effects are cumulative. For example,

```
cc -CLIST:linelength=120:show=ON -TARG:platform=ip27 -TARG:isa=mips4
```

Many of the grouped options are documented in this and the following chapters. All compiler options mentioned are in the index under the heading "compiler option."

Compiler Defaults

The compiler has defaults for all options, but often they are not what you might expect. For example, the compiler optimizes the generated code for a particular hardware type as specified with the *-TARG* option group (detailed later in "Setting Target System with -TARG" on page 95). The default for the target hardware is not, as you might expect, the system on which the compiler runs, but the POWER CHALLENGE R10000—as of release 7.2.1. The default might change in a later release.

You can alter the compiler's choice of defaults for some, but not all, options by changing the file */etc/compiler.defaults*, as documented in the *MIPSpro Compiling and Performance Tuning Guide* and the cc(1) documents listed in "Related Documents" on page xxix. However, this changes the defaults for all compilations by all users on that system.

Because at least some defaults are in the control of the administrator of the system, you cannot be certain of the compiler's defaults on any given system. As a result, the only way to be certain of consistent results is to specify all important compiler options on every compile. The only convenient way of doing this is with a Makefile.

Using a Makefile

You want to use the many compiler options consistently, regardless of which computer you use to run the compilation; also, you want to experiment with alternative options in a controlled way. The best way to avoid confusion and to get consistent results is to use a Makefile in which you define all important compiler options as variables. Example C-4 on page 292 shows a starting point for a Makefile.

For a straightforward program, a Makefile like the one in Example C-4 may be all you need. To use it, create a directory for your program and move the source files into the directory. Create a file named Makefile containing the lines of Example C-4. Edit the file and fill in the lines that define EXEC (the program name) and FOBJS and COBJS, the names of the component object files.

To build the program, enter

```
% make
```

The *smake* program compiles the source modules into object modules using the compiler flags specified in the variables defined in the first few lines, then links the executable program.

To remove the executable and objects, so as to force recompilation of all files, use

```
% make clean
```

You can override the Makefile setting of any of the variables by specifying its value on the command line. For example, if the OLEVEL variable does not permit debugging (*-Ofast=ip27* does not), you can compile a debugging version with the following commands:

```
% make clean; make OLEVEL="-O0 -g3"
```

You can add extra compilation options with the DEFINES variable. For example,

```
% make DEFINES="-DDEBUG -OPT:alias=restrict"
```

Setting Optimization Level with -On

The basic optimization flag is *-On*, where *n* is 0, 1, 2, 3, or *fast*. This flag controls which optimizations the compiler will attempt: the higher the optimization level, the more aggressive the optimizations that will be tried. In general, the higher the optimization level, the longer compilation takes.

Start with -O2 for All Modules

A good beginning point for program tuning is optimization level *-O2* (or, equivalently, *-O*). This level performs extensive optimizations that are conservative; that is, they will not cause the program to have numeric roundoff characteristics different from an unoptimized program. Sophisticated (and time-consuming) optimizations such as software pipelining and loop nest optimizations are not performed.

In addition, the compiler does not rearrange the sequence of code very much at this level. At higher levels, the compiler can rearrange code enough that the correspondence between a source line and the generated code becomes hazy. If you profile a program compiled at the *-O2* level, you can make use of a *prof -heavy* report (see "Including Line-Level Detail" on page 73). When you compile at higher levels, it can be difficult to interpret a profile because of code motion and inlining.

Use the *-O2* version of your program as the baseline for performance. For some programs, this may be the fastest version. The higher optimization levels have their greatest effects on loop-intensive code and math-intensive code. They may bring little or no improvement to programs that are strongly oriented to integer logic and file I/O.

Compile -O3 or -Ofast for Critical Modules

You should identify the program modules that consume a significant fraction of the program's run time and compile them with the highest level of optimization. This may be specified with either *-O3* or *-Ofast*. Both flags enable software pipelining and loop nest optimizations. The difference between the two is that *-Ofast* includes additional optimizations:

- Interprocedural analysis (discussed later under "Exploiting Interprocedural Analysis" on page 125).
- Arithmetic rearrangements that can cause numeric roundoff to be different from an unoptimized program (this can be controlled separately, see "Understanding Arithmetic Standards" on page 95).
- Assumption that certain pointer variables are independent, not aliased (also controlled separately; see "Understanding Aliasing Models" on page 109).

With no argument, *-Ofast* assumes the default target for execution (see "Compiler Defaults" on page 92). You can specify which machine will run the generated code by naming the "ip" number of the CPU board. The complete list of valid board numbers is given in the cc(1) reference page. The ip number of any system is displayed by the command *hinv -c processor*.For all SN0 systems, use *-Ofast=ip27*.

Use -O0 for Debugging

Use the lowest optimization level, *-O0*, when debugging your program. This flag turns off all optimizations, so there is a direct correspondence between the original source code and the machine instructions the compiler generates.

You can run a program compiled to higher levels of optimization under a symbolic debugger, but the debugger will have trouble showing you the program's progress statement by statement, because the compiler often merges statements and moves code around. An optimized program, under a debugger, can be made to stop on procedure entry, but cannot reliably be made to stop on a specified statement.

-O0 is the default when you don't specify an optimization level, so be sure to specify the optimization level you want.

Setting Target System with -TARG

Although the R10000, R8000, and R5000 CPUs all support the MIPS IV ISA (see "Selecting an ABI and ISA" on page 46), their internal hardware implementations are quite different.

The compiler does not simply choose the machine instructions to be executed, which would be the same for any MIPS IV CPU. It carefully schedules the sequence of instructions so that the CPU will make best use of its parallel functional units. The ideal sequence is different for each of the CPUs that support MIPS IV. Code scheduled for one chip runs correctly, but not necessarily in optimal time, on another.

You can instruct the compiler to generate code optimized for a particular chip via the *-r10000*, *-r8000*, and *-r5000* flags. These flags, when used while linking, cause libraries optimized for the specified chip, if available, to be linked. Specially optimized versions of *libfastm* exist for some CPUs.

You can also specify the target chip using an alternative flag, *-TARG:proc=rN*. This flag tells the compiler to generate code optimized for the specified chip. However, the flag is ignored at link time, so it does not affect which libraries are linked. If *-r10000* has been used, *-TARG:proc=r10000* is unnecessary.

In addition to specifying the target chip, you can also specify the target processor board. This gives the compiler information about hardware parameters such as cache sizes, which can be used in some optimizations. You can specify the target platform with the *-TARG:platform=ipxx* flag. For SN0, use *-TARG:platform=ip27*.

The *-Ofast=ip27* flag implies the flags *-r10000*, *-TARG:proc=r10000*, and *-TARG:platform=ip27*.

Understanding Arithmetic Standards

For some applications, control of numeric error is paramount, and the programmer devotes great care to ordering arithmetic operations so that the finite precision of computer registers will not introduce and propagate errors. The IEEE 754 standard for floating-point arithmetic is conservatively designed to preserve accuracy, at the expense of speed if necessary, and conformance to it is a key element of a numerically-sensitive program—especially a program that must produce identical answers in different environments.

Many other programs and programmers are less concerned about preserving every bit of precision and more interested in getting answers that are in adequate agreement, as fast as possible. Two compiler options control these issues.

The nuances of mathematical accuracy are discussed in depth in the math(3) reference page. The compiler options discussed in this section are detailed in the opt(5) reference page.

IEEE Conformance

The *-OPT:IEEE_arithmetic* option controls how strictly the generated code adheres to the IEEE 754 standard for floating-point arithmetic. The flexibility exists for a couple of reasons:

- The MIPS IV ISA defines hardware instructions (such as reciprocal and reciprocal square root) that do not meet the accuracy specified by IEEE 754 standard. Although the inaccuracy is small (no more than two units in the last place, or 2 ulp), a program that uses these instructions is not strictly IEEE-compliant, and might generate a result that is different from a compliant program.
- Some standard optimization techniques also produce different results from those of a strictly IEEE-compliant implementation.

For example, a standard optimization would be to lift the constant expression out of the following loop:

```
do i = 1, n
   b(i) = a(i)/c
enddo
```

This results in an optimized loop as follows:

```
tmp = 1.0/c
do i = 1, n
   b(i) = a(i)*tmp
enddo
```

This is not allowed by the IEEE standard, because multiplying by the reciprocal (even a completely IEEE-accurate representation of the reciprocal) in place of division can produce results that differ in the least significant place.

You use the compiler flag *-OPT:IEEE_arithmetic=n* to tell the compiler how to handle these cases. The flag is documented (along with some other IEEE-related options) in the opt(5) reference page. Briefly, the three allowed values of *n* mean:

1 Full compliance with the standard. Use only when full compliance is essential.

2 Permits use of the slightly-inexact MIPS IV hardware instructions.

3 Permits the compiler to use any identity that is algebraically valid, including rearranging expressions.

Unless you know that numerical accuracy is a priority for your program, start with level 3. If numerical instability seems to be a problem, change to level 2, so that the compiler will not rearrange your expressions. Use level 1 only when IEEE compliance is a priority.

A related flag is *-OPT:IEEE_NaN_inf*. When this flag specified as OFF, which it is by default, the compiler is allowed to make certain assumptions that the IEEE standard forbids, for example, assuming that x/x is 1.0 without performing the division.

Roundoff Control

The order in which arithmetic operations are coded into a program dictates the order in which they are carried out. The Fortran code in Example 5-1 specifies that the variable *sum* is calculated by first adding *a(1)* to *sum*, and to that result is added *a(2)*, and so on until *a(n)* has been added in.

Example 5-1 Simple Summation Loop

```
sum = 0.0
do i = 1, n
   sum = sum + a(i)
enddo
```

In most cases, including Example 5-1, the programmed sequence of operations is not necessary to produce a numerically valid result. Insisting on the programmed sequence can keep the compiler from taking advantage of the hardware resources.

For example, suppose the target CPU has a pipelined floating point add unit with a two-cycle latency. When the compiler can rearrange the code, it can unroll the loop, as shown in Example 5-2.

Example 5-2 Unrolled Summation Loop

```
sum1 = 0.0
sum2 = 0.0
do i = 1, n-1, 2
   sum1 = sum1 + a(i)
   sum2 = sum2 + a(i+1)
enddo
do i = i, n
   sum1 = sum1 + a(i)
enddo
sum = sum1 + sum2
```

The compiler unrolls the loop to put two additions in each iteration. This reduces the index-increment overhead by half, and also supplies floating-point adds in pairs so as to keep the CPU's two-stage adder busy on every cycle. (The final wind-down loop finishes the summation when *n* is odd.)

The final result in *sum* is algebraically identical to the result produced in Example 5-1. However, because of the finite nature of computer arithmetic, these two loops can produce answers that differ in the final bits—bits that should be considered roundoff error places in any case.

The *-OPT:roundoff* flag allows you to specify how strictly the compiler must adhere to such coding-order semantics. As with the *-OPT:IEEE_arithmetic* flag above, there are several levels that are detailed in reference page opt(5). Briefly, these levels are:

0 Requires strict compliance with programmed sequence, thereby inhibiting most loop optimizations.

1 Allows simple reordering within expressions.

2 Allows extensive reordering like the unrolled reduction loop in Example 5-2. Default when -O3 is specified.

3 Allows any algebraically-valid transformation. Default when -Ofast is used.

Start with level 3 in a program in which floating-point computations are incidental, otherwise with level 2.

Roundoff level three permits the compiler to transform expressions, and it also implies the use of inline machine instructions for float-to-integer conversion (separately controlled with the *-OPT:fast_nint=on* and *-OPT:fast-trunc=on*).These transformations can cause the roundoff characteristic of a program to change. For most programs, small differences in roundoff are not a problem, but in some instances the program's results can

be different enough from what it produced previously to be considered wrong. That event might call into question the numeric stability of the algorithm, but the easiest course of action may be to use a lower roundoff level in order to generate the expected results.

If you are using *-Ofast* and want to reduce the roundoff level, you will need to set the desired level explicitly:

```
% cc -n32 -mips4 -Ofast=ip27 -OPT:roundoff=2 ...
```

Exploiting Software Pipelining

Loops imply parallelism. This ranges from, at minimum, instruction-level parallelism up to parallelism sufficient to carry out different iterations of the loop on different CPUs. Multiprocessor parallelism will be dealt with later; this section concentrates on instruction-level parallelism.

Understanding Software Pipelining

Consider the loop in Example 5-3, which implements the vector operation conventionally called DAXPY.

Example 5-3 Basic DAXPY Loop

```
do i = 1, n
   y(i) = y(i) + a*x(i)
enddo
```

This adds a vector multiplied by a scalar to a second vector, overwriting the second vector. Each iteration of the loop requires the following instructions:

- Two loads (of x(i) and y(i))
- One store (of y(i))
- One multiply-add
- Two address increments
- One loop-end test
- One branch

The superscalar architecture of the CPU can execute up to four instructions per cycle. These time-slots can be filled by any combination of the following:

- One load or store operation
- One ALU 1 instruction
- One ALU 2 instruction
- One floating-point add
- One floating-point multiply

In addition, most instructions are pipelined, and have associated pipeline latencies ranging from one cycle to dozens. Software pipelining is a compiler technique for optimally filling the superscalar time slots. The compiler figures out how best to schedule the instructions to fill up as many superscalar slots as possible, in order to run the loop at top speed.

In software pipelining, each loop iteration is broken up into instructions, as was done in the preceding paragraphs for DAXPY. The iterations are then overlapped in time in an attempt to keep all the functional units busy. Of course, this is not always possible. For example, a loop that does no floating point will not use the floating-point adder or multiplier. And some units are forced to remain idle on some cycles because they have to wait for instructions with long pipeline latencies, or because only four instructions can be selected from the above menu. In addition, the compiler has to generate code to fill the pipeline on the first and second iterations, and code to wind down the operation on the last iteration, as well as special-case code to handle the event that the loop requires only 1 or 2 iterations.

The central part of the generated loop, meant to be executed many times, executes with the pipeline full and tries to keep it that way. The compiler can determine the minimum number of cycles required for this steady-state portion of the loop, and it tries to schedule the instructions so that this minimum-time performance is approached as the number of iterations in a loop increases.

In calculating the minimum number of cycles required for the loop, the compiler assumes that all data are in cache (specifically, in the primary data cache). If the data are not in cache, the loop takes longer to execute. (In this case, the compiler can generate additional instructions to *prefetch* data from memory into the cache before they are needed. See “Understanding Prefetching” on page 137.) The remainder of this discussion of software pipelining considers only in-cache data. This may be unrealistic, but it is easier to determine if the software pipeline steady-state has achieved the peak rate the loop is capable of.

Pipelining the DAXPY Loop

To see how effective software pipelining is, let us return to the DAXPY example (Example 5-2 on page 98)and consider its instruction schedule. The CPU is capable of, at most, one memory instruction and one multiply-add per cycle. Because three memory operations are done for each multiply-add, the optimal performance for this loop is one-third of the peak floating-point rate of the CPU.

In creating an instruction schedule, the compiler has to consider many hardware restrictions. The following is a sample of the restrictions that affect the DAXPY loop:

- Only one load or store may be performed on each CPU cycle.
- A load of a floating-point datum from the primary cache has a latency of three cycles.
- However, a madd instruction does a multiply followed by an add, and the load of the addend to the instruction can begin just one cycle before the madd, because two cycles will be expended on the multiply before the addend is needed.
- It takes four cycles to complete a madd when the result is passed to another floating point operation, but five cycles when the output must be written to the register file, as is required to store the result.
- The source operand of a store instruction is accessed two cycles after the address operand, so the store of the result of a madd instruction can be started 5 - 2 = 3 cycles after the madd begins.

Based on these considerations, the best schedule for an iteration of the basic DAXPY loop (Example 5-3 on page 99) is shown in Table 5-2:

Table 5-2 Instruction Schedule for Basic DAXPY

Cycle	Inst 1	Inst 2	Inst 3	Inst 4
0	ld *x	++x		
1	ld *y			
2				
3				madd
4				
5				
6	st *y	br	++y	

The simple schedule for this loop achieves two floating-point operations (multiply and add, in one instruction) each seven cycles, or one-seventh of the potential peak.

This schedule can be improved only by unrolling the loop so that it calculates two vector elements per iteration, as shown in Example 5-4.

Example 5-4 Unrolled DAXPY Loop

```
do i = 1, n-1, 2
   y(i+0) = y(i+0) + a*x(i+0)
   y(i+1) = y(i+1) + a*x(i+1)
enddo
do i = i, n
   y(i+0) = y(i+0) + a*x(i+0)
enddo
```

The best schedule for the steady-state part of the unrolled loop is shown in Table 5-3

Table 5-3 Instruction Schedule for Unrolled DAXPY

Cycle	Inst 1	Inst 2	Inst 3	Inst 4
0	ld *(x+0)			
1	ld *(x+1)			
2	ld *(y+0)	x+=2		
3	ld *(y+1)			madd0
4				madd1
5				
6	st *(y+0)			
7	st *(y+1)	y+=2	br	

This loop achieves four floating point operations in eight cycles, or one-fourth of peak; it is closer to the nominal rate for this loop, one-third of peak, but still not perfect.

A compiler-generated software pipeline schedule for the DAXPY loop is shown in Example 5-5.

Example 5-5 Compiler-Generated DAXPY Schedule

```
[ windup code to load registers omitted ]
loop:
        Cycle  0:       ld y4
        Cycle  1:       st y0
        Cycle  2:       st y1
        Cycle  3:       st y2
        Cycle  4:       st y3
        Cycle  5:       ld x5
        Cycle  6:       ld y5
        Cycle  7:       ld x6
        Cycle  8:       ld x7                   madd4
        Cycle  9:       ld y6                   madd5
        Cycle 10:       ld y7   y+=4            madd6
        Cycle 11:       ld x0           bne out madd7
        Cycle  0:       ld y0
        Cycle  1:       st y4
        Cycle  2:       st y5
        Cycle  3:       st y6
        Cycle  4:       st y7
        Cycle  5:       ld x1
        Cycle  6:       ld y1
        Cycle  7:       ld x2
        Cycle  8:       ld x3                   madd0
        Cycle  9:       ld y2                   madd1
        Cycle 10:       ld y3   y+=4            madd2
        Cycle 11:       ld x4           beq loop madd3
out:
[ winddown code to store final results omitted]
```

This schedule is based on unrolling the loop four times. It achieves eight floating point operations in twelve cycles, or one-third of peak, so its performance is optimal. This schedule consists of two blocks of 12 cycles. Each 12-cycle block is called a replication since the same code pattern repeats for each block. Each replication accomplishes four iterations of the loop, containing four madds and 12 memory operations, as well as a pointer increment and a branch, so each replication achieves the nominal peak calculation rate. Iterations of the loop are spread out over the replications in a pipeline so that the superscalar slots are filled up.

Figure 5-1 shows the memory operations and madds for 12 iterations of the DAXPY loop. Time, in cycles, runs horizontally across the diagram.

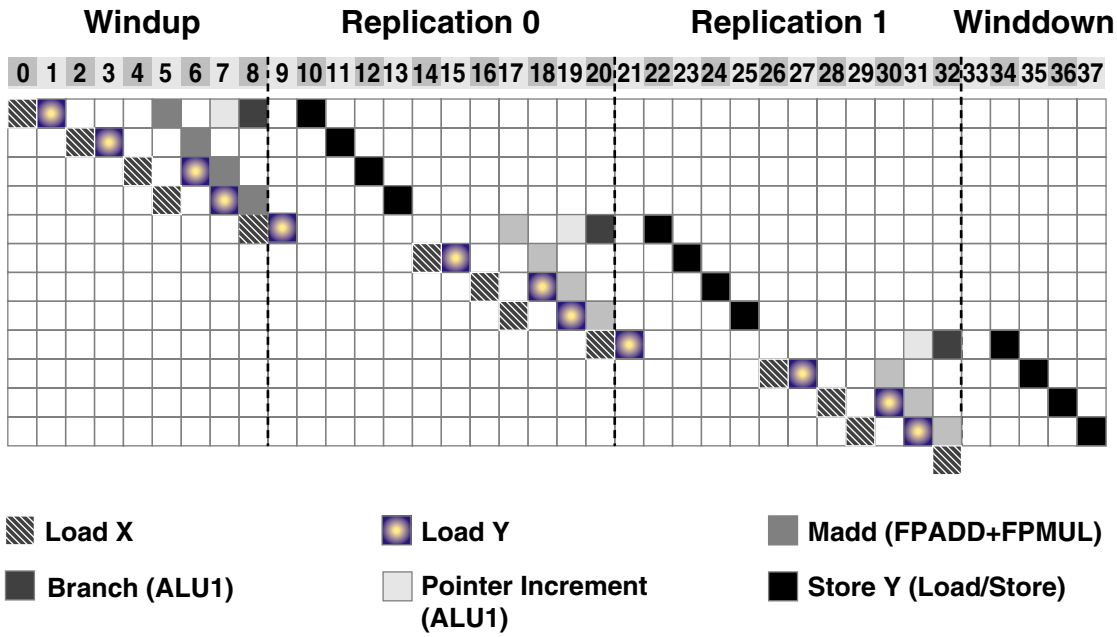


Figure 5-1 DAXPY Software Pipeline Schedule

The first five iterations fill the software pipeline. This is the windup stage; it was omitted from Example 5-5 for brevity. Cycles 9 through 20 make up the first replication; cycles 21 through 32, the second replication; and so on. The compiler has unrolled the loop four times, so each replication completes four iterations. Each iteration is spread out over several cycles: 15 for the first of the four unrolled iterations, 10 for the second, and 9 for each of the third and fourth iterations. Because only one memory operation can be executed per cycle, the four iterations must be offset from one another, and it takes 18 cycles to complete all four of them. Two replications are required to contain the full 18 cycles of work for the four iterations.

A test at the end of each iteration determines if all possible groups of four iterations have been started. If not, execution drops through to the next replication or branches back to the first replication at the beginning of the loop. If all groups of four iterations have begun, execution jumps out to code that finishes the in-progress iterations. This is the wind-down code; for brevity it was omitted from Example 5-5.

Reading Software Pipelining Messages

To help you determine how effective the compiler has been at software pipelining, the compiler can generate a report card for each loop. You can request that these report cards be written to a listing file, or assembler output can be generated to see them.

As an example, once again consider the DAXPY loop, as in Example 5-6.

Example 5-6 Basic DAXPY Loop Code

```
subroutine daxpy(n,a,x,y)
integer n
real*8 a, x(n), y(n)
do i = 1, n
   y(i) = y(i) + a*x(i)
enddo
return
end
```

The software pipelining report card is produced as part of the assembler file that is an optional output of a compilation. Use the *-S* compiler flag to generate an assembler file; the report card will be mixed in with the assembly code, as follows (the flag *-LNO:opt=0* prevents certain cache optimizations that would obscure the current study of software pipelining):

```
% f77 -n32 -mips4 -O3 -OPT:IEEE_arithmetic=3 -LNO:opt=0 -S daxpy.f
```

An assembly listing is voluminous and confusing. You can use the shell script *swplist* (shown in Example C-5 on page 294), which generates the assembler listing, extracts the report cards, and inserts each into a source listing file just above the loop it pertains to. The listing file is given the same name as the file compiled, but with the extension *.swp*. *swplist* is used with the same options one uses to compile:

```
% swplist -n32 -mips4 -O3 -OPT:IEEE_arithmetic=3 -LNO:opt=0 -c daxpy.f
```

The report card generated by either of these methods resembles Example 5-7.

Example 5-7 Sample Software Pipeline Report Card

```
#<swps>
#<swps> Pipelined loop line 6 steady state
#<swps>
#<swps>    50 estimated iterations before pipelining
#<swps>     2 unrollings before pipelining
#<swps>     6 cycles per 2 iterations
#<swps>     4 flops        ( 33% of peak) (madds count as 2)
#<swps>     2 flops        ( 16% of peak) (madds count as 1)
#<swps>     2 madds        ( 33% of peak)
#<swps>     6 mem refs     (100% of peak)
#<swps>     3 integer ops  ( 25% of peak)
#<swps>    11 instructions ( 45% of peak)
#<swps>     2 short trip threshold
#<swps>     7 ireg registers used.
#<swps>     6 fgr registers used.
#<swps>
```

The report card tells you how effectively the CPU's hardware resources are being used in the schedule generated for the loop. There are lines showing how many instructions of each type there are per replication and what percentage of each resource's peak utilization has been achieved.

The report card in Example 5-7 is for the loop starting at line 6 of the source file (this line number may be approximate). The loop was unrolled two times and then software pipelined. The schedule generated by the compiler requires six cycles per replication, and each replication completes two iterations because of the unrolling.

The steady-state performance of the loop is 33% of the floating-point peak. The "madds count as 2" line counts floating-point operations. This indicates how fast the CPU can run this loop on an absolute scale. The peak rate of the CPU is two floating-point operations per cycle, or twice the CPU clock rate. Because this loop reaches 33% of peak, it has an absolute rate of two-thirds the CPU clock, or 133 MFLOPS in a 200 MHz CPU.

Alternatively, the "madds count as 1" line counts floating-point instructions. Since the floating point work in most loops does not consist entirely of madd instructions, this count indicates how much of the floating point capability of the chip is being utilized in the loop given its particular mix of floating point instructions. A 100% utilization here, even if the floating point operation rate is less than 100% of peak, means that the chip is running as fast as it can in this loop.

For the R10000 CPU, which has an adder unit and a multiplier, this line does not provide much additional information beyond the "madds count as 2" line. In fact, the utilization percentage is misleading in this example, because both of the floating-point instructions in the replication are actually madds, and these use both the adder and the multiplier. Here, the instruction is counted as if it simply uses one of the units. On an R8000 CPU—for which there are two complete floating point units, each capable of add, multiply, and madd instructions—this line provides better information. You can easily see cases in which the chip's floating point units are 100% utilized even though the CPU is not sustaining two floating-point operations per unit per cycle.

Each software pipeline replication described in Example 5-7 contains six memory references, the most possible in this many cycles (shown as "100% of peak"). Thus, this loop is memory-constrained. Sometimes that's unavoidable, and no additional (non-memory) resources can make the loop run faster. But in other situations, loops can be rewritten to reduce the number of memory operations, and in these cases performance can be improved. An example of such a loop is presented later.

Example 5-7 shows that only three integer operations are used (presumably a branch and two pointer increments). The total number of instructions for the replication is 11, so there are plenty of free superscalar slots—the theoretical limit would be 24, or 4 per cycle.

The report card indicates that there is a "short trip threshold" of two. This means that, for loops with fewer than two groups of two iterations, a separate, non-pipelined piece of code is used. Building up and draining the pipeline imposes some overhead, so pipelining is avoided for very short loops.

Finally, Example 5-7 reports that seven integer registers (ireg) and six floating-point registers (fgr) were required by the compiler to schedule the code. Availability of registers is not an issue in this case, but bigger and more complicated loops require more registers, sometimes more than the CPU provides. In these cases, the compiler must either introduce some scratch memory locations to hold intermediate results, increasing the number of load and store operations for the loop, or stretch out the loop to allow greater reuse of the existing registers. When these problems occur, splitting the loop into smaller pieces can often eliminate them.

This report card provides you with valuable information. If a loop is running well, that is validated by the information it contains. If the speed of the loop is disappointing, it provides clues as to what should be done—either via the use of additional compiler flags or by modifying the code—to improve the performance.

Enabling Software Pipelining with -O3

The compiler does not software pipeline loops by default. To enable software pipelining, you must use the highest level of optimization, *-O3*. This can be specified with either *-Ofast*, as above, or with *-O3*; the difference is that *-Ofast* is a superset of *-O3* and also turns on other optimizations such as interprocedural analysis (discussed later under "Exploiting Interprocedural Analysis" on page 125). (You can use the switch *-SWP:=ON* with lower levels of optimization, but *-O3* also enables loop nest and other optimizations that complement software pipelining, so this alternative is not recommended.)

Software pipelining is not done by default, because it increases compilation time. Although finding the number of cycles in an optimal schedule is easy, producing a schedule given a finite number of registers is difficult. Heuristic algorithms are employed to try to find a schedule. In some instances they fail, and only a schedule with more cycles than optimal can be generated. The schedule that is found can vary depending on subtle code differences or the choice of compiler flags.

Dealing with Software Pipelining Failures

Some loops are well suited to software pipelining. In general, vectorizable loops, such as the DAXPY loop (see Example 5-5) compile well with no special effort from you. But getting some loops to software pipeline can be difficult. Fortunately, other optimizations—interprocedural analysis, loop nest optimizations—and compiler directives can help in these situations.

For an in-order superscalar CPU such as the R8000 CPU, the key loops in a program must be software pipelined to achieve good performance; the functional units sit idle if concurrent work is not scheduled for them. With its out-of-order execution, the R10000 CPU can be much more forgiving of the code generated by the compiler; even code scheduled for another CPU can still run well. Nevertheless, there is a limit to the instruction reordering the CPU can accomplish, so getting the most time-consuming loops in a program to software pipeline is still key to achieving top performance.

Not all loops software pipeline. One common cause of this is a subroutine or function call inside the loop. When a loop contains a call, the software pipelining report card is:

```
#<swpf> Loop line 6 wasn't pipelined because:
#<swpf>       Function call in the DoBody
#<swpf>
```

If the called subroutine contains loops of its own, then this software pipelining failure is inconsequential because it is the loops in the subroutine that need to be pipelined. If there are no loops in the subroutine, however, finding a way to get the loop containing it to software pipeline may be important for achieving good performance.

One way to remedy this is to replace the subroutine call with a copy of the subroutine's code. This is *inlining*, and although it can be done manually, the compiler can do it automatically as part of interprocedural analysis ("Exploiting Interprocedural Analysis" on page 125).

While-loops, and loops that contain goto statements that jump out of the loop, do not software pipeline. Non-pipelined loops can still run well on SN0 systems. For in-order CPUs such as the R8000, however, such loops could incur a significant performance penalty, so if an alternate implementation exists that converts the loop to a standard do- or for-loop, use it.

Finally, sometimes the compiler notes a "recurrence" in the SWP report card. This indicates that the compiler believes that the loop depends, or might depend, on a value produced in a preceding iteration of the loop. This kind of dependency between iterations inhibits pipelining. Sometimes it is real; more often it is only apparent, and can be removed by specifying a different aliasing model.

Informing the Compiler

Optimization level *-O3* enables a variety of optimizations including software pipelining and loop nest optimization. But a handful of other options and directives can improve the software pipelined schedules that the compiler generates and speed up non-pipelined code as well. These are described next, before delving into fine-tuning interprocedural analysis and loop nest optimizations.

Understanding Aliasing Models

The problem with a pointer is that it might point to anything, including a named variable or the value also addressed by a different pointer. The possibility of "aliasing"—the possibility that two unrelated names might refer to the identical memory—creates uncertainty that requires the compiler to make the most conservative assumptions about dependencies caused by dereferencing pointer variables.

Without assurance from the programmer, the compiler needs to be cautious and assume that any two pointers can point to the same location in memory. This caution removes many optimization opportunities, particularly for loops. When this is not the case, you can tell the compiler using the *-OPT:alias=* flag. The many possible aliasing models are detailed in the opt(5) reference page. The following table summarizes their meanings:

any	Any pointer can point to anything. C default.
common_scalar	A scalar in a common block cannot be aliased as part of an array in the block.
cray_pointer	Pointers never refer to named variables, other restrictions.
no_cray_pointer	Pointer storage can overlay named variables. Fortran default.
typed	Pointers of distinct types point to distinct objects. Default when -Ofast is used.
no_typed	Pointers of different types can point to the same memory.
unnamed	Pointers never refer to named variables.
no_unnamed	Pointers can point to named variables.
restrict	Distinct pointers point to distinct, non-overlapping memory.
no_restrict	Distinct pointers can point to overlapping memory areas.
parm	Parameters do not alias variables. Fortran default.
no_parm	Parameters can alias variables.

Use Alias=Restrict When Possible

When you are sure that different pointer variables never point to overlapping regions of memory, use the *-OPT:alias=restrict* to allow the compiler to generate the best-performing code. If you use this flag to compile code in which different pointers *can* alias the same data, incorrect code can be generated. That would not be a compiler problem; you provided the compiler with false information.

As an example, consider the C implementation of DAXPY in Example 5-8.

Example 5-8 C Implementation of DAXPY Loop

```
void
daxpy(int n, double a, double *x, double *y)
{
   int i;
   for (i=0; i<n; i++) {
      y[i] += a * x[i];
   }
}
```

When Example 5-8 is compiled with the following command, it produces the software pipelining message is as shown in Example 5-9:

```
% cc -n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3 -LNO:opt=0
        -c daxpy.c
```

(The flag *-LNO:opt=0* is used to prevent certain cache optimizations that would obscure this study of software pipelining performance.)

Example 5-9 SWP Report Card for C Loop with Default Alias Model

```
#<swps> Pipelined loop line 8 steady state
#<swps>
#<swps>   100 estimated iterations before pipelining
#<swps>       Not unrolled before pipelining
#<swps>     7 cycles per iteration
#<swps>     2 flops        ( 14% of peak) (madds count as 2)
#<swps>     1 flop         (  7% of peak) (madds count as 1)
#<swps>     1 madd         ( 14% of peak)
#<swps>     3 mem refs     ( 42% of peak)
#<swps>     3 integer ops  ( 21% of peak)
#<swps>     7 instructions ( 25% of peak)
#<swps>     1 short trip threshold
#<swps>     4 ireg registers used.
#<swps>     3 fgr registers used.
#<swps>     3 min cycles required for resources
#<swps>     6 cycles required for recurrence(s)
#<swps>     3 operations in largest recurrence
#<swps>     7 cycles in minimum schedule found
```

The compiler cannot determine the data dependencies between *x* and *y,* and has to assume that storing into *y[i]* could modify some element of *x*. The compiler plays it safe and produces a schedule that is more than twice as slow as is ideal.

When *-OPT:alias=restrict* is used, the report resembles Example 5-10.

Example 5-10 SWP Report Card for C Loop with Alias=Restrict

```
#<swps> Pipelined loop line 8 steady state
#<swps>
#<swps>    50 estimated iterations before pipelining
#<swps>     2 unrollings before pipelining
#<swps>     6 cycles per 2 iterations
#<swps>     4 flops        ( 33% of peak) (madds count as 2)
#<swps>     2 flops        ( 16% of peak) (madds count as 1)
#<swps>     2 madds        ( 33% of peak)
#<swps>     6 mem refs     (100% of peak)
#<swps>     3 integer ops  ( 25% of peak)
#<swps>    11 instructions ( 45% of peak)
#<swps>     2 short trip threshold
#<swps>     7 ireg registers used.
#<swps>     6 fgr registers used.
```

Because it knows that the *x* and *y* arrays are nonoverlapping, the compiler can generate an ideal schedule. As long as this subroutine has been specified so that this assumption is correct, the code is valid and runs significantly faster.

Use Alias=Disjoint When Necessary

When the code uses pointers to pointers, *-OPT:alias=restrict* is insufficient to tell the compiler that data dependencies do not exist. If this is the case, using *-OPT:alias=disjoint* allows the compiler to ignore potential data dependencies (*disjoint* is a superset of *restrict*).

C code using multidimensional arrays is a good candidate for the *-OPT:alias=disjoint* flag. Consider the nest of loops shown in Example 5-11.

Example 5-11 C Loop Nest on Multidimensional Array

```
for ( i=1; i< size_x1-1 ;i++)
  for ( j=1; j< size_x2-1 ;j++)
     for ( k=1; k< size_x3-1 ;k++)
      {
```

```
      out3 = weight3*(    field_in[i-1][j-1][k-1]
                        + field_in[i+1][j-1][k-1]
                        + field_in[i-1][j+1][k-1]
                        + field_in[i+1][j+1][k-1]
                        + field_in[i-1][j-1][k+1]
                        + field_in[i+1][j-1][k+1]
                        + field_in[i-1][j+1][k+1]
                        + field_in[i+1][j+1][k+1] );
      out2 = weight2*(    field_in[i  ][j-1][k-1]
                        + field_in[i-1][j  ][k-1]
                        + field_in[i+1][j  ][k-1]
                        + field_in[i  ][j+1][k-1]
                        + field_in[i-1][j-1][k  ]
                        + field_in[i+1][j-1][k  ]
                        + field_in[i-1][j+1][k  ]
                        + field_in[i+1][j+1][k  ]
                        + field_in[i  ][j-1][k+1]
                        + field_in[i-1][j  ][k+1]
                        + field_in[i+1][j  ][k+1]
                        + field_in[i  ][j+1][k+1] );
      out1 = weight1*(    field_in[i  ][j  ][k-1]
                        + field_in[i  ][j  ][k+1]
                        + field_in[i  ][j+1][k  ]
                        + field_in[i+1][j  ][k  ]
                        + field_in[i-1][j  ][k  ]
                        + field_in[i  ][j-1][k  ] );
      out0 = weight0*     field_in[i  ][j  ][k  ];
      field_out[i][j][k] = out0+out1+out2+out3;
    }
```

Here, the three-dimensional arrays are declared as

```
double*** field_out;
double*** field_in;
```

Without input from the user, the compiler cannot tell if there are data dependencies between different elements of one array, let alone both. Because of the multiple levels of indirection, *-OPT:alias=restrict* does not provide sufficient information. As a result, compiling Example 5-11 with the following options produces the following software pipeline report card shown in Example 5-12:

```
% cc -n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3:alias=restrict
    -c stencil.c
```

Example 5-12 SWP Report Card for Stencil Loop with Alias=Restrict

```
#<swps> Pipelined loop line 77 steady state
#<swps>
#<swps>   100 estimated iterations before pipelining
#<swps>       Not unrolled before pipelining
#<swps>    42 cycles per iteration
#<swps>    30 flops        ( 35% of peak) (madds count as 2)
#<swps>    27 flops        ( 32% of peak) (madds count as 1)
#<swps>     3 madds        (  7% of peak)
#<swps>    42 mem refs     (100% of peak)
#<swps>    16 integer ops  ( 19% of peak)
#<swps>    85 instructions ( 50% of peak)
#<swps>     1 short trip threshold
#<swps>    20 ireg registers used.
#<swps>    12 fgr registers used.
```

However, you can instruct the compiler that even multiple indirections are independent with *-OPT:alias=disjoint*. Compiling with this aliasing model produces the results shown in Example 5-13.

Example 5-13 SWP Report Card for Stencil Loop with Alias=Disjoint

```
#<swps>
#<swps> Pipelined loop line 77 steady state
#<swps>
#<swps>   100 estimated iterations before pipelining
#<swps>       Not unrolled before pipelining
#<swps>    28 cycles per iteration
#<swps>    30 flops        ( 53% of peak) (madds count as 2)
#<swps>    27 flops        ( 48% of peak) (madds count as 1)
#<swps>     3 madds        ( 10% of peak)
#<swps>    28 mem refs     (100% of peak)
#<swps>    12 integer ops  ( 21% of peak)
#<swps>    67 instructions ( 59% of peak)
#<swps>     2 short trip threshold
#<swps>    14 ireg registers used.
#<swps>    15 fgr registers used.
#<swps>
```

This is a 40% improvement in performance (28 cycles per iteration, down from 42), and all it took was a flag to tell the compiler that the pointers can't alias each other.

Breaking Other Dependencies

In some situations, code ambiguity goes beyond pointer aliasing. Consider the loop in Example 5-14.

Example 5-14 Indirect DAXPY Loop

```
void idaxpy(
int     n,
double  a,
double  *x,
double  *y,
int     *index)
{
  int   i;
  for (i=0; i<n; i++) {
    y[index[i]] += a * x[index[i]];
  }
}
```

You have seen this DAXPY operation before, except here the vector elements are chosen indirectly through an index array. (The same technique could as easily be used in Fortran.) Assume that the vectors x and y do not overlap; and compile as before (again with *-LNO:opt=0* only to clarify the results).

```
% cc -n32 -mips4 -Ofast=ip27 -OPT:IEEE_arithmetic=3:alias=restrict
        -LNO:opt=0 -c idaxpy.c
```

The results, as shown in Example 5-15, are less than optimal.

Example 5-15 SWP Report Card on Indirect DAXPY

```
 #<swps>    50 estimated iterations before pipelining
 #<swps>     2 unrollings before pipelining
 #<swps>    10 cycles per 2 iterations
 #<swps>     4 flops        ( 20% of peak) (madds count as 2)
 #<swps>     2 flops        ( 10% of peak) (madds count as 1)
 #<swps>     2 madds        ( 20% of peak)
 #<swps>     8 mem refs     ( 80% of peak)
 #<swps>     8 integer ops  ( 40% of peak)
 #<swps>    18 instructions ( 45% of peak)
 #<swps>     1 short trip threshold
 #<swps>    10 ireg registers used.
 #<swps>     4 fgr registers used.
 #<swps>
 #<swps>     8 min cycles required for resources
 #<swps>    10 cycles in minimum schedule found
```

When run at optimal speed, this loop should be constrained by loads and stores. In that case, the memory references would be at 100% of peak. The problem is that, since the values of the *index* array are unknown, there could be data dependencies between loop iterations for the vector *y*. For example, if *index[i]* = 1 for all *i*, all iterations would update the same vector element. In general, if the values in *index* are not a permutation of unique integers from 0 to *n*-1 (1 to *n*, in Fortran), it could happen that a value stored on one iteration could be referenced on the next. In that case, the best pipelined code could produce a wrong answer.

If you know that the values of the array *index* are such that no dependencies will occur, you can inform the compiler of this via the ivdep compiler directive. For a Fortran program, the directive looks like Example 5-16.

Example 5-16 Indirect DAXPY in Fortran with ivdep

```
c$dir ivdep
   do i = 1, n
      y(ind(i)) = y(ind(i)) + a * x(ind(i))
   end
```

In C, the directive is a pragma and is used as shown in Example 5-17.

Example 5-17 Indirect DAXPY in C with ivdep

```
#pragma ivdep
   for (i=0; i<n; i++) {
      y[index[i]] += a * x[index[i]];
   }
```

With this change, the schedule is improved by 25%, as shown in Example 5-18.

Example 5-18 SWP Report Card for Indirect DAXPY with ivdep

```
#<swps>    50 estimated iterations before pipelining
#<swps>     2 unrollings before pipelining
#<swps>     8 cycles per 2 iterations
#<swps>     4 flops        ( 25% of peak) (madds count as 2)
#<swps>     2 flops        ( 12% of peak) (madds count as 1)
#<swps>     2 madds        ( 25% of peak)
#<swps>     8 mem refs     (100% of peak)
#<swps>     8 integer ops  ( 50% of peak)
#<swps>    18 instructions ( 56% of peak)
#<swps>     2 short trip threshold
#<swps>    14 ireg registers used.
#<swps>     4 fgr registers used.
```

The ivdep pragma gives the compiler permission to ignore the dependencies that are possible from one iteration to the next. As long as those dependencies do not actually occur, the code produces correct answers. But if they do in fact occur, the output can be wrong. Just as when misusing the alias models, this is a case of user error.

There are a few subtleties in how the ivdep directive works. First, note that the directive applies only to inner loops. Second, it applies only to dependencies that are carried by the loop. Consider the code segment in Example 5-19.

Example 5-19 Loop with Two Types of Dependency

```
do i= 1, n
   a(index(1,i)) = b(i)
   a(index(2,i)) = c(i)
enddo
```

Two types of dependencies are possible in this loop. If *index(1,i)* is the same as *index(1,i+k)*, or if *index(2,i)* is the same as *index(2,i+k)*, for some value of *k* such that *i+k* is in the range 1 to *n*, there is a loop-carried dependency. Because the code is in a loop, some element of *a* will be stored into twice, inappropriately. (If the loop is parallelized, you cannot predict which assignment will take effect.)

If *index(1,i)* is the same as *index(2,i)* for some value of *i*, there is a non-loop-carried dependency. Some element of *a* will be assigned from both *b* and *c*, and this would happen whether the code was in a loop or not.

In Example 5-17 there is a potential loop-carried dependency. The ivdep directive informs the compiler that the dependency is not actually present—there are no duplicate values in the index array. But if the compiler detects a potential non-loop-carried dependency (a dependency between statements in one iteration), the ivdep directive does not instruct the compiler to ignore these potential dependencies. To force even these dependencies to be overlooked, extend the behavior of the ivdep directive via the *-OPT:liberal_ivdep=TRUE* flag. When this flag is used, an ivdep directive means to ignore all dependencies in a loop.

With either variant of ivdep, the compiler issues a warning when the directive has instructed it to break an obvious dependence.

Example 5-20 C Loop with Obvious Loop-Carried Dependence

```
for (i=0; i<n; i++) {
   a[i] = a[i-1] + 3;
}
```

In Example 5-20 the dependence is real. Forcing the compiler to ignore it almost certainly produces incorrect code. Nevertheless, an ivdep pragma breaks the dependence, and the compiler issues a warning and then schedules the code so that the iterations are carried out independently.

One more variant of ivdep is enabled with the flag *-OPT:cray_ivdep=TRUE*. This instructs the compiler to use the semantics that Cray Research specified when it originated this directive for its vectorizing compilers. That is, only "lexically forward and backward" loop-carried dependencies are broken. Example 5-20 is an instance of a lexically backwards dependence. Example 5-21 is not.

Example 5-21 C Loop with Lexically-Forward Dependency

```
for (i=0; i<=n; i++) {
   a[i] = a[i+1] + 3;
}
```

Here, the dependence is from the load (*a[i+1]*) to the store (*a[i]*), and the load comes lexically before the store. In this case, a Cray semantics ivdep directive would not break the dependence.

Improving C Loops

The freedom of C looping syntax, as compared to Fortran loops, presents the compiler with some challenges. You provide extra information by specifying an alias model and the ivdep pragma, and these allow you to achieve good performance from C despite the language semantics. But a few other things can be done to improve the speed of programs written in C.

Global variables can sometimes cause performance problems, because it is difficult for the compiler to determine whether they alias other variables. If loops involving global variables do not software pipeline as efficiently as they should, you can often cure the problem by making local copies of the global variables.

Another problem arises from using dereferenced pointers in loop tests.

Example 5-22 C Loop Test Using Dereferenced Pointer

```
for (i=0; i<(*n); i++) {
   y[i] += a * x[i];
}
```

The loop in Example 5-22 will not software pipeline, because the compiler cannot tell whether the value addressed by *n* will change during the loop. If you know that this value is loop-invariant, you can inform the compiler by creating a temporary scalar variable to use in the loop test, as shown in Example 5-23.

Example 5-23 C Loop Test Using Local Copy of Dereferenced Pointer

```
int   lim = *n;
for (i=0; i<lim; i++) {
   y[i] += a * x[i];
}
```

The loop in Example 5-23 should now pipeline comparably to Example 5-10.

Example 5-24 shows a different case of loop-invariant pointers leading to aliasing problems.

Example 5-24 C Loop with Disguised Invariants

```
void sub1(
   double  **dInputToHidden,
   double   *HiddenDelta,
   double   *Input,
   double  **InputToHidden,
   double    alpha,
   double    eta,
   int NumberofInputs,
   int NumberofHidden)
{
   int i, j;
   for (i=0; i<NumberofInputs; i++) {
      for (j=0; j<NumberofHidden; j++) {
         dInputToHidden[i][j] = alpha * dInputToHidden[i][j]
                              + eta * HiddenDelta[j] * Input[i];
      InputToHidden[i][j] += dInputToHidden[i][j];
      }
   }
}
```

This loop is similar to a DAXPY; after pipelining it should be memory bound with about a 50% floating-point utilization. But compiling with *-OPT:alias=disjoint* does not quite achieve this level of performance.

Example 5-25 SWP Report Card for Loop with Disguised Invariance

```
#<swps> Pipelined loop line 19 steady state
#<swps>   100 estimated iterations before pipelining
#<swps>       Not unrolled before pipelining
#<swps>     7 cycles per iteration
#<swps>     5 flops        ( 35% of peak) (madds count as 2)
#<swps>     4 flops        ( 28% of peak) (madds count as 1)
#<swps>     1 madd         ( 14% of peak)
#<swps>     7 mem refs     (100% of peak)
#<swps>     6 integer ops  ( 42% of peak)
#<swps>    17 instructions ( 60% of peak)
#<swps>     3 short trip threshold
#<swps>    16 ireg registers used.
#<swps>     8 fgr registers used.
```

The problem is the pointers indexed by *i*; they cause the compiler to see dependencies that aren't there. This problem can be fixed by hoisting the invariants out of the inner loop, as shown in the emphasized statements in Example 5-26.

Example 5-26 C Loop with Invariants Exposed

```
void sub2(
   double  **dInputToHidden,
   double   *HiddenDelta,
   double   *Input,
   double  **InputToHidden,
   double    alpha,
   double    eta,
   int NumberofInputs,
   int NumberofHidden)
{
   int i, j;
   double *dInputToHiddeni, *InputToHiddeni, Inputi;
   for (i=0; i<NumberofInputs; i++) {
      dInputToHiddeni = dInputToHidden[i];
      InputToHiddeni = InputToHidden[i];
      Inputi = Input[i];
      for (j=0; j<NumberofHidden; j++) {
         dInputToHiddeni[j] = alpha * dInputToHiddeni[j]
                            + eta * HiddenDelta[j] * Inputi;
         InputToHiddeni[j] += dInputToHiddeni[j] ;
      }
   }
}
```

When the loop is written this way, the compiler can see that *dInputToHidden[i]*, *InputToHidden[i]*, and *Input[i]* will not change during execution of the inner loop, so it can schedule that loop more aggressively, resulting in 5 cycles per iteration rather than 7.

Example 5-27 SWP Report Card for Modified Loop

```
#<swps> Pipelined loop line 52 steady state
#<swps>   100 estimated iterations before pipelining
#<swps>       Not unrolled before pipelining
#<swps>     5 cycles per iteration
#<swps>     5 flops        ( 50% of peak) (madds count as 2)
#<swps>     4 flops        ( 40% of peak) (madds count as 1)
#<swps>     1 madd         ( 20% of peak)
#<swps>     5 mem refs     (100% of peak)
#<swps>     4 integer ops  ( 40% of peak)
#<swps>    13 instructions ( 65% of peak)
#<swps>     3 short trip threshold
#<swps>     8 ireg registers used.
#<swps>    10 fgr registers used.
```

Permitting Speculative Execution

When extra information is conveyed to the compiler, it can generate faster code. One useful bit of information that the compiler can sometimes exploit is knowledge of which hardware exceptions can be ignored. An exception is an event that requires intervention by the operating system. Examples include floating-point exceptions such as divide by zero, and memory access exceptions such as segmentation fault. Any program can choose which exceptions it will pay attention to and which it will ignore. For example, floating-point underflows generate exceptions, but their effect on the numeric results is usually negligible, so they are typically ignored by flushing them to zero (via hardware). Divide by zero is generally enabled because it usually indicates a fatal error in the program.

Software Speculative Execution

How the compiler takes advantage of disabled exceptions is best seen through an example. Consider a program that is known not to make invalid memory references. It runs fine whether memory exceptions are enabled or disabled. But if they are disabled and the compiler knows this, it can generate code that may speculatively make references to potentially invalid memory locations; for example, a reference to an array element that might lie beyond the end of an array. Such code can run faster than nonspeculative code

on a superscalar architecture, if it takes advantage of functional units that would otherwise be idle. In addition, the compiler can often reduce the amount of software pipeline wind-down code when it can make such speculative memory operations; this allows more of the loop iterations to be spent in the steady-state code, leading to greater performance.

When the compiler generates speculative code, it moves instructions in front of a branch that previously had protected them from causing exceptions. For example, accesses to memory elements beyond the end of an array are normally protected by the test at the end of the loop. Speculative code can be even riskier than this. For instance, a source line such as that in Example 5-28 might be compiled into the equivalent of Example 5-29.

Example 5-28 Conventional Code to Avoid an Exception

```
if (x .ne. 0.0) b = 1.0/x
```

Example 5-29 Speculative Equivalent Permitting an Exception

```
tmp = 1.0/x
if (x .ne. 0.0) b = tmp
```

In Example 5-29, the division operation is started first so that it can run in the floating point unit concurrently with the test of *x*. When *x* is nonzero, the time to perform the test essentially disappears. When *x* is zero, the invalid result of dividing by zero is discarded, and again no time is lost. However, division by zero is no longer protected by the branch; an exception can occur but should be ignored.

Allowing such speculation is the only way to get some loops to software pipeline. On an in-order superscalar CPU such as the R8000, speculation is critical to achieving top performance on loops containing branches. To allow this speculation to be done more freely by the compiler, most exceptions are disabled by default on R8000 based systems.

Hardware Speculative Execution

For the R10000 CPU, however, the hardware itself performs speculative execution. Instructions can be executed out of order. The CPU guesses which way a branch will go, and speculatively executes the instructions that follow on the predicted path. Executing code equivalent to Example 5-28, the R10000 CPU is likely to predict that the branch will find *x* nonzero, and to speculatively execute the division before the test is complete. When *x* is zero, an exception results that would not normally occur in the program. The CPU dismisses exceptions caused by a speculative instruction without operating system intervention. Thus, most exceptions can be enabled by default in R10000-based systems.

Owing to the hardware speculation in the R10000 CPU, software speculation by the compiler is less critical from a performance point of view. Moreover, software speculation can cause problems, because exceptions encountered on a code path generated by the compiler are not simply dismissed—unlike speculative exceptions created by the hardware. In the best case, the program wastes time handling software traps.

Controlling the Level of Speculation

For these reasons, when generating code for the R10000 CPU, the compiler by default only performs the safest speculative operations. Which operations it speculates on are controlled by the *-TENV:X=n* flag, where *n* has the following meanings, as described in the cc(1) or f77(1) reference page:

0 No speculative code movement.

1 Speculation that might cause IEEE-754 underflow and inexact exceptions.

2 Speculation that might cause any IEEE-754 exceptions except zero-divide.

3 Speculation that might cause any IEEE-754 exceptions.

4 Speculation that might cause any exception including memory access.

The compiler generates code to disable software handling of any exception that is permitted by the *-TENV:X* flag.

The default is *-TENV:X=1* for optimization levels of *-O2* or below, and *-TENV:X=2* for *-O3* or *-Ofast*. Normally you should use the default values. Only when you have verified that the program never generates exceptions (see “Using Exception Profiling” on page 81), should you use higher values, which sometimes provide a modest increase in performance.

A program that divides by zero in course of the execution are not correct and should be aborted with a divide by zero exception. Compiling such a program with *-TENV:X=3* or *-TENV:X=4* will continue execution after the division by zero, with the effect that NaN and Inf values will propagate through the calculations and show up in the results.

Division by zero generated by the compiler for optimization will be safely discarded under these circumstances and will not propagate. However, the compiler generated exceptions cannot be distinguished from the genuine program errors, which makes it diffucult to trap on the program errors (see “Understanding Treatment of Underflow Exceptions” on page 81) when *TENV:X=3* is used.

The runtime library assumes the most liberal exception setting, so programs linked with libraries compiled X=3 will have the division by zero exception disabled. This is currently the case with the scientific libraries SCSC and complib.sgimath. The fpe mechanism can still be used to trap errors. If any exceptions are generated by the libraries (as seen with the trace back) they are just an artifact of the compiler optimization and do not need to be analyzed.

Passing a Feedback File

The compiler tries to schedule looping and branching code for the CPU so that the CPU's speculative execution will be most effective. When scheduling code, the question arises: how often will a branch be taken? The compiler has only the source code, so it can only schedule branches based on heuristics (that is, on educated guesses). An example of a heuristic is the guess that the conditional branch that closes a loop is taken much more often than not. Another is that the then-part of an if is used more often than it is skipped.

As described under "Creating a Compiler Feedback File" on page 75, you can create a file containing exact counts of the use of branches during a run of a program. Then you can feed this file to the compiler when you next compile the program. The compiler uses these counts to control how it schedules code around conditional branches.

After you create a feedback file, you pass it to the compiler using the *-fb filename* option. You would pass the feedback file for the entire program when compiling any module of that program, or you can pass it for the compilation of only the critical modules. The feedback information is optional, and the compiler ignores it when the statement numbers in the feedback file do not match the source code as it presently exists.

One difficulty with feedback files is that they must be created from a non-optimized executable. When the executable program is compiled with optimization, the simple relationship between executable code and source statements is lost. If you compile with *-fb* and the compiler issues a warning about mismatched statement numbers, the cause is that the feedback file came from a trace of an optimized program.

The proper sequence to make use a feedback file is as follows:

- Compile the entire program with *-O0* optimization.
- Use *ssrun -ideal* to take an ideal profile of the unoptimized program.
- Use *prof -feedback* to create the feedback file from this profile.
- Recompile the entire program with full optimization and *-fb*.

This rather elaborate process is best reserved for the final compilation before releasing a completed program.

Exploiting Interprocedural Analysis

Most compiler optimizations work within a single routine (function or procedure) at a time. This helps keep the problems manageable and is a key aspect of supporting separate compilation because it allows the compiler to restrict attention to the current source file.

However, this narrow focus also presents serious restrictions. Knowing nothing about other routines, the optimizer is forced to make worst-case assumptions about the possible effects of calls to them. For instance, at any function call it must usually generate code to save and then restore the state of all variables that are in registers.

Interprocedural analysis (IPA) analyzes more than a single routine—preferably the entire program—at once. It can transfer information across routine boundaries, and these results are available to subsequent optimization phases, such as loop nest optimization and software pipelining. The optimizations performed by the MIPSpro compilers's IPA facility include the following:

- Inlining: replacing a call to a routine with a modified copy of the routine's code, when the called routine is in a different source file.
- Constant propagation: When a global variable is initialized to a constant value and never modified, each use of its name is replaced by the constant value. When a parameter of a routine is passed as a particular constant value in every call, the parameter is replaced in the routine's code by the constant.
- Common block array padding: Increasing the dimensions of global arrays in Fortran to reduce cache conflicts.
- Dead function elimination: Functions that are never called can be removed from the program image, improving memory utilization.

- Dead variable elimination: Variables that are never used can be eliminated, along with any code that initializes them.
- PIC optimization: finding names that are declared external, but which can safely be converted to internal or hidden names. This allows simpler, faster function-call protocols to be used, and reduces the number of slots need in the global table.
- Global name optimizations: Global names in shared code must normally be referenced via addresses in a global table, in case they are defined or preempted by another DSO. If the compiler knows it is compiling a main program and that the name is defined in another of the source files making up the main program, an absolute reference can be substituted, eliminating a memory reference.
- Automatic global table sizing: IPA can optimize the data items that can be referenced by simple offsets from the GP register, instead of depending on the programmer to provide an ideal value through the *-G* compiler option.

The first two optimizations, inlining and constant propagation, have the effect of enabling further optimizations. Inlining removes function calls from loops (helping to eliminate dependencies) while giving the software pipeliner more statements to work with in scheduling. After constant propagation, the optimizer finds more common subexpressions to combine, or finds more invariant expressions that it can lift out of loops.

Requesting IPA

IPA is easy to use. Use an optimization level of *-O2* or *-O3* and add *-IPA* to both the compile and link steps when building your program. Alternatively, use *-Ofast* for both compile and link steps; this optimization flag enables IPA.

Compiling and Linking with IPA

Unlike other IPA implementations, the MIPSpro compilers do not require that all the components of a program be present and compiled at once for IPA analysis. IPA works by postponing much of the compilation process until the link step, when all the program components can be analyzed together.

The compile step, applied to one or more source modules, does syntax analysis and places an intermediate representation of the code into the *.o* output file, instead of normal relocatable code. This intermediate code, called WHIRL code in some compiler documentation, contains preliminary IPA and optimization information (this is why all compiler flags must be given at the compile step).

When all source files have been compiled into *.o* files, the program is linked as usual with a call to the *ld* command (or to *cc* or *f77*, which in turn call *ld*). The linker recognizes that at least some of its input object files are incompletely-compiled WHIRL files. The link step can include object files (*.o*) and archives (*.a*) that have been compiled without IPA analysis, as well as referencing DSOs that, however they were compiled, cannot contribute detailed information because a DSO may be replaced with a different version after the program is built.

The linker invokes IPA, which analyzes and transforms all the input WHIRL objects, writing modified versions of the objects. The linker then invokes the compiler back end on each of the modified objects, so that it can complete the optimization phases and generate code to produce a normal relocatable object. Finally, when all input files have been reduced to object code, the linker can proceed to link the object files.

If there are modules that you would prefer to compile using an optimization level of *-O1* or *-O0*, that's fine. Without IPA, a module is compiled to a relocatable object file. They will link into the program with no problems, but no interprocedural analysis will be performed on them. The linker accepts a combination of WHIRL and object files, and IPA is applied as necessary.

However, all modules that are compiled with *-IPA* should also be compiled to the identical optimization level—normally *-O3*, or *-Ofast* which implies both *-O3* and *-IPA*, but you can compile modules with *-O2* and *-IPA*. The problem is that if the compiler inlines code into a module compiled *-O2*, the inlined code is not software pipelined or otherwise optimized, even though it comes from a module compiled *-O3*.

Compile Time with IPA

When IPA is used, the proportion of time spent in compiling and linking changes. The compile step is usually faster with IPA, because the majority of the optimizations are postponed until the link step. However, the link step takes longer, because it is now responsible for most of the compilation.

The total build time is likely to increase for two reasons: First, although the IPA step itself may not be expensive, inlining usually increases the size of the code, and therefore expands the time required by the rest of the compiler. Second, because IPA analysis can propagate information from one module into optimization decisions into many other modules, even minor changes to a single component module cause most of the program compilation to be redone. As a result, IPA is best suited for use on programs with a stable source base.

Understanding Inlining

One key feature of IPA is inlining, the replacement of a call to a routine with the code of that routine. can improve performance in several ways:

- By eliminating the subroutine call, the overhead in making the call, such as saving and restoring registers, is also eliminated.
- Possible aliasing effects between routine parameters and global variables are eliminated.
- A larger amount of code is exposed to later optimization phases of the compiler. So, for example, loops that previously could not be software pipelined now can be.

But there are some disadvantages to inlining:

- Inlining expands the size of the source code and so increases compilation time.
- The resulting object text is larger, taking more space on disk and more time to load.
- The additional object code takes more room in the cache, sometimes causing more primary and secondary cache misses.

Thus, inlining does not always improve program execution time. As a result, there are flags you can use to control which routines will be inlined.

Inlining is available in Fortran 77, C, and C++ with the 7.2 compiler release; it will be part of Fortran 90 in a subsequent release of the compilers. There are a few restrictions on inlining. The following types of routines cannot be inlined:

- Recursive routines.
- Routines using static (SAVE in Fortran) local variables.
- Routines with variable argument lists (varargs).
- Calls in which the caller and callee have mismatched argument lists.
- Routines written in one language cannot be inlined into a module written in a different language.

IPA supports two types of inlining: manual and automatic. The manual (also known as standalone) inliner is invoked with the *-INLINE* option group. It has been provided to support the C++ inline keyword but can also be used for C and Fortran programs using the command-line option. It inlines only the routines that you specify and they must be located in the same source file as the routine into which they are inlined. The automatic inliner is invoked whenever *-IPA* (or *-Ofast*) is used. It uses heuristics to decide which routines will be beneficial to inline, so you don't need to specify these explicitly. It can inline routines from any files that have been compiled with *-IPA*.

Using Manual Inlining

The manual inliner is controlled with the *-INLINE* option group. Use a combination of the following four options to specify which routines to inline:

-INLINE:all	Inline all routines except those specified to *-INLINE:never*.
-INLINE:none	Inline no routines except those specified to *-INLINE:must*.
-INLINE:must=*name,...*	Specify a list of routines to inline at every call.
-INLINE:never=*name,...*	Specify a list of routines never to inline.

Consider the code in Example 5-30.

Example 5-30 Code Suitable for Inlining

```
      subroutine zfftp ( z, n, w, s )
      real*8 z(0:2*n-1), w(2*(n-1)), s(0:2*n-1)
      integer n
      integer iw, i, m, l, j, k
      real*8 w1r, w1i, w2r, w2i, w3r, w3i
      iw = 1
c     First iteration of outermost loop over l&m: n = 4*lm.
      i = 0
      m = 1
      l = n/4
      if (l .gt. 1) then
         do j = 0, l-1
            call load4(w(iw),iw,w1r,w1i,w2r,w2i,w3r,w3i)
            call radix4pa(z(2*(j+0*l)),z(2*(j+1*l)),
     &                    z(2*(j+2*l)),z(2*(j+3*l)),
     &                    s(2*(4*j+0)),s(2*(4*j+1)),
     &                    s(2*(4*j+2)),s(2*(4*j+3)),
     &                    w1r,w1i,w2r,w2i,w3r,w3i)
         enddo
         i = i + 1
         m = m * 4
         l = l / 4
      endif
c ...
      return
      end
```

The loop in Example 5-30 contains two subroutine calls whose code is shown in Example 5-31.

Example 5-31 Subroutine Candidates for Inlining

```
      subroutine load4 ( w, iw, w1r, w1i, w2r, w2i, w3r, w3i )
      real*8 w(0:5), w1r, w1i, w2r, w2i, w3r, w3i
      integer iw
c
      w1r = w(0)
      w1i = w(1)
      w2r = w(2)
      w2i = w(3)
      w3r = w(4)
      w3i = w(5)
      iw = iw + 6
      return
      end
c-----------------------------------------------------------------------
      subroutine radix4pa ( c0, c1, c2, c3, y0, y1, y2, y3,
     &                      w1r, w1i, w2r, w2i, w3r, w3i )
      real*8 c0(0:1), c1(0:1), c2(0:1), c3(0:1)
      real*8 y0(0:1), y1(0:1), y2(0:1), y3(0:1)
      real*8 w1r, w1i, w2r, w2i, w3r, w3i
c
      real*8 d0r, d0i, d1r, d1i, d2r, d2i, d3r, d3i
      real*8 t1r, t1i, t2r, t2i, t3r, t3i
c
      d0r = c0(0) + c2(0)
      d0i = c0(1) + c2(1)
      d1r = c0(0) - c2(0)
      d1i = c0(1) - c2(1)
      d2r = c1(0) + c3(0)
      d2i = c1(1) + c3(1)
      d3r = c3(1) - c1(1)
      d3i = c1(0) - c3(0)
      t1r = d1r + d3r
      t1i = d1i + d3i
      t2r = d0r - d2r
      t2i = d0i - d2i
      t3r = d1r - d3r
      t3i = d1i - d3i
      y0(0) = d0r + d2r
      y0(1) = d0i + d2i
```

```
      y1(0) = w1r*t1r - w1i*t1i
      y1(1) = w1r*t1i + w1i*t1r
      y2(0) = w2r*t2r - w2i*t2i
      y2(1) = w2r*t2i + w2i*t2r
      y3(0) = w3r*t3r - w3i*t3i
      y3(1) = w3r*t3i + w3i*t3r
      return
      end
```

Both subroutines are straight-line code. Having this code isolated in subroutines undoubtedly makes the source code easier to read and to maintain. However, inlining this code reduces function call overhead; more important, inlining allows the calling loop to software pipeline.

If all the code is contained in the file *fft.f*, you can inline the calls with the following compilation command:

```
% f77 -n32 -mips4 -O3 -r10000 -OPT:IEEE_arithmetic=3:roundoff=3
       -INLINE:must=load4,radix4pa -flist -c fft.f
```

(Here, *-O3* is used instead of *-Ofast* because the latter would invoke *-IPA* to inline the subroutines automatically.) The *-flist* flag tells the compiler to write out a Fortran file showing what transformations the compiler has done. This file is named *fft.w2f.f* ("w2f" stands for WHIRL-to-Fortran). You can confirm that the subroutines were inlined by checking the w2f file. It contains the code shown in Example 5-32.

Example 5-32 Inlined Code from w2f File

```
      l = (n / 4)
      IF(l .GT. 1) THEN
        DO j = 0, l + -1, 1
          w1i = w((j * 6) + 2)
          w2r = w((j * 6) + 3)
          w2i = w((j * 6) + 4)
          w3r = w((j * 6) + 5)
          w3i = w((j * 6) + 6)
          d0r = (z((j * 2) + 1) + z((((l * 2) + j) * 2) + 1))
          d0i = (z((j * 2) + 2) + z((((l * 2) + j) * 2) + 2))
          d1r = (z((j * 2) + 1) - z((((l * 2) + j) * 2) + 1))
          d1i = (z((j * 2) + 2) - z((((l * 2) + j) * 2) + 2))
          d2r = (z(((l + j) * 2) + 1) + z((((l * 3) + j) * 2) + 1))
          d2i = (z(((l + j) * 2) + 2) + z((((l * 3) + j) * 2) + 2))
          d3r = (z((((l * 3) + j) * 2) + 2) - z(((l + j) * 2) + 2))
          d3i = (z(((l + j) * 2) + 1) - z((((l * 3) + j) * 2) + 1))
          t1r = (d1r + d3r)
          t1i = (d1i + d3i)
```

```
      t2r = (d0r - d2r)
      t2i = (d0i - d2i)
      t3r = (d1r - d3r)
      t3i = (d1i - d3i)
      s((j * 8) + 1) = (d0r + d2r)
      s((j * 8) + 2) = (d0i + d2i)
      s((j * 8) + 3) = ((w((j * 6) + 1) * t1r) -(w1i * t1i))
      s((j * 8) + 4) = ((w((j * 6) + 1) * t1i) +(w1i * t1r))
      s((j * 8) + 5) = ((w2r * t2r) -(w2i * t2i))
      s((j * 8) + 6) = ((w2r * t2i) +(w2i * t2r))
      s((j * 8) + 7) = ((w3r * t3r) -(w3i * t3i))
      s((j * 8) + 8) = ((w3r * t3i) +(w3i * t3r))
    END DO
  ENDIF
  RETURN
```

There are two other ways to confirm that the routines have been inlined:

1. Use the option *-INLINE:list* to instruct the compiler to print out the name of each routine as it is being inlined.
2. Check the performance to verify that the loop software pipelines (you cannot check the assembler file because the *-S* option disables IPA).

Using Automatic Inlining

Automatic inlining is a feature of IPA. The compiler performs inlining on must subroutine calls, but no longer considers inlining to be beneficial when it increases the size of the program by 100% or more, or if the extra code causes the *-Olimit* option to be exceeded. This options specifies the maximum size, in basic blocks, of a routine that the compiler will optimize. Generally, you do not need to worry about this limit because, if it is exceeded, the compiler will print a warning message instructing you to recompile using the flag *-Olimit=nnnn*.

If one of these limits is exceeded, IPA will restrict the inlining rather than not do any at all. It decides which routines will remain inlined on the basis of their location in the call hierarchy. Routines at the lowest depth in the call graph are inlined first, followed by routines higher up in the hierarchy. This is illustrated in Figure 5-2.

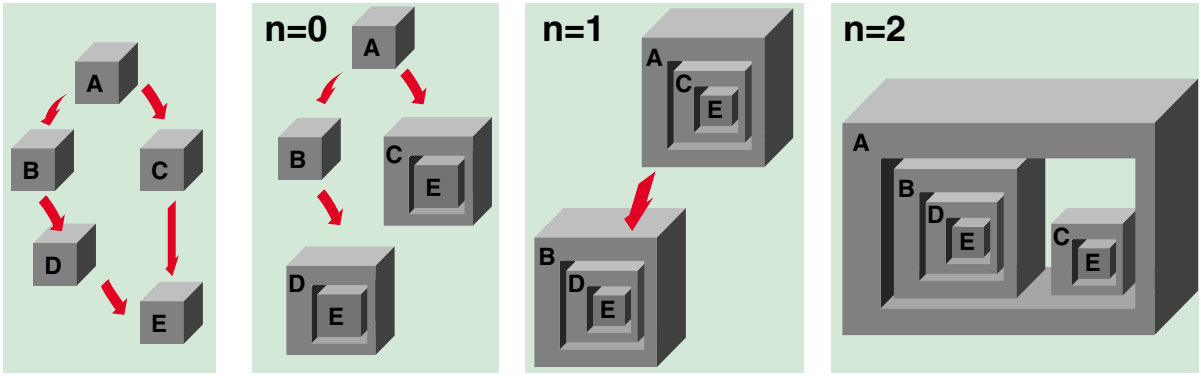


Figure 5-2 Inlining and the Call Hierarchy

The call tree for a small program is shown on the left. At the lowest depth, leaf routines (those that do not call any other routine) are inlined. In Figure 5-2, the routine E is inlined into routines D and C. At the next level, D is inlined into B and C is inlined into A. At the last level, which for this program is the top level, B is inlined into A. IPA stops automatic inlining at the level that causes the program size to double, or the level that exceeds the *-Olimit* option.

You can increase or decrease the amount of automatic inlining with these three options:

-IPA:space=*n*	The largest acceptable percentage increase in code size. The default 100%.
-IPA:forcedepth=*n*	All routines at depth *n* or less in the call graph must be inlined, regardless of the space limit.
-IPA:inline=off	Turns off automatic inlining altogether.

Automatic inlining inlines the calls to **load4** and **radix4pa** in Example 5-30. But because most of IPA's actions are not performed until the link stage, you don't get a *.w2f* file with which to verify this until the link has been completed.

The *-INLINE:must* and *-INLINE:never* flags work in conjunction with automatic inlining. To block automatic inlining of specific routines, name them in an *-INLINE:never* list. To require inlining of specific routines regardless of code size or their position in the call graph, name them in an *-INLINE:must* list.

IPA Programming Hints

The *-IPA* option, whether given explicitly on compile and link lines or implicitly via *-Ofast*, is designed to work fairly automatically. Nevertheless, you can do a few things to enhance interprocedural analysis.

IPA needs to know which routines are being called in order to do its analysis, so it is limited by programming constructs that obscure the call graph. The use of function pointers, virtual functions, and C++ exception handling are all examples of things that limit interprocedural analysis.

The optimizations that IPA performs other than inlining should always benefit performance. Although inlining can often significantly improve performance, it can also hurt it, so it pays to experiment with the flags available for controlling automatic inlining.

Summary

This chapter has toured the concepts of SWP and IPA, and shown how these optimizations are constrained by the compiler's need to always generate code that is correct under the worst-case assumptions, with the result that opportunities for faster code are often missed. By giving the compiler more information, and sometimes by making minor adjustments in the source code, you can get the tightest possible machine code for the innermost loops.

The code schedule produced by pipelining does not take memory access delays into account. Cache misses and virtual page faults can destroy the timing of any code schedule. Optimizing access to memory takes place at a higher level of optimization, discussed in the next chapter.

Chapter 6

Optimizing Cache Utilization

Because the SN0 architecture has multiple levels of memory access, optimizations that improve a program's utilization of memory can have major effects. This chapter introduces the concepts of cache and virtual memory use and its effects on performance in these topics:

- "Understanding the Levels of the Memory Hierarchy" on page 135 describes the relation between the programmer's view of memory as arrays and variables, and the system's view of memory as a hierarchy of caches and nodes.
- "Identifying Cache Problems with Perfex and SpeedShop" on page 142 shows how you can tell when a program is being slowed by memory access problems.
- "Using Other Cache Techniques" on page 148 describes some basic strategies you can use to overcome memory-access problems.

The discussion continues in Chapter 7, "Using Loop Nest Optimization," with details of the compiler optimizations that automatically exploit the memory hierarchy.

Understanding the Levels of the Memory Hierarchy

To understand why cache utilization problems occur, it is important to understand how the data caches in the system function. As mentioned under "Cache Architecture" on page 22, the CPUs in the SN0 architecture utilize a two-level cache hierarchy: there is a 32 KB L1 cache on the CPU chip, and a much larger L2 cache external to the CPU on the node board. The virtual memory system makes the third and fourth levels of caching.

Understanding Level-One and Level-Two Cache Use

The L1 and L2 caches are two-way set associative; that is, any data element may reside in only two possible locations in either cache, as determined by the low-order bits of its address. If the program refers to two locations that have the same low-order address bits, copies of both can be maintained in the cache simultaneously. When a third datum with the same low-order address bits is accessed, it replaces one of the two previous data—the one that was least recently accessed.

To maintain a high bandwidth between the caches and memory, each location in the cache holds a contiguous block, or *cache line*, of data. For the primary cache, the line size is 32 bytes; it is 128 bytes for the secondary cache. Thus, when a particular word is copied into a cache, several nearby words are copied with it.

The CPU can make use only of data in registers, and it can load data into registers only from the primary cache. So data must be brought into the primary cache before it can be used in calculations. The primary cache can obtain data only from the secondary cache, so all data in the primary cache is simultaneously resident in the secondary cache. The secondary cache obtains its data from main memory. The operating system uses main memory to create a larger virtual address space for each process, keeping unneeded pages of data on disk and loading them as required—so, in effect, main memory is a cache also, with a cache line size of 16KB, the size of a virtual page.

A *cache miss* occurs when the program refers to a datum that is not present in the cache. That instruction is suspended while the datum (and the rest of the cache line that contains it) is copied into the cache. When an instruction refers to a datum that is already in the cache, this is a cache hit.

Understanding TLB and Virtual Memory Use

A page is the smallest contiguous block of memory that the operating system allocates to your program. Memory is divided into pages to allow the operating system to freely partition the physical memory of the system among the running processes while presenting a uniform virtual address space to all jobs. Your programs see a contiguous range of virtual memory addresses, but the physical memory pages can be scattered all over the system.

The operating system translates between the virtual addresses your programs use and the physical addresses that the hardware requires. These translations are done for every memory access, so, to keep their overhead low, the 64 most recently used page addresses are cached in the *translation lookaside buffer (TLB)*. This allows virtual-to-physical translation to be carried out with zero latency, in hardware—for those 64 pages.

When the program refers to a virtual address that is not cached in the TLB, a TLB miss has occurred. Only then must the operating system assist in address translation. The hardware traps to a routine in the IRIX kernel. It locates the needed physical address in a memory table and loads it into one of the TLB registers, before resuming execution.

Thus the TLB acts as a cache for frequently-used page addresses. The virtual memory of a program is usually much larger than 64 pages. The most recently used pages of memory (hundreds or thousands of them) are retained in physical memory, but depending on program activity and total system use, some virtual pages can be copied to disk and removed from memory. Sometimes, when a program suffers a TLB miss, the operating system discovers that the referenced page is not presently in memory—a "page fault" has occurred. Then the program is delayed while the page is read from disk.

Degrees of Latency

The *latency* of data access becomes greater with each cache level. Latency of memory access is best measured in CPU clock cycles. One cycle occupies from 4 to 6 nanoseconds, depending on the CPU clock speed. The latencies to the different levels of the memory hierarchy are as follows:

- CPU Register: 0 cycles.
- L1 cache hit: 2 or 3 cycles.
- L1 cache miss satisfied by L2 cache hit: 8 to 10 cycles.
- L2 cache miss satisfied from main memory, no TLB miss: 75 to 250 cycles; that is, 300 to 1100 nanoseconds, depending on the node where the memory resides (see Table 1-3 on page 19).
- TLB miss requiring only reload of the TLB to refer to a virtual page already in memory: approximately 2000 cycles.
- TLB miss requiring virtual page to load from backing store: hundreds of millions of cycles; that is, tens to hundreds of milliseconds.

A miss at each level of the memory hierarchy multiplies the latency by an order of magnitude or more. Clearly a program can sustain high performance only by achieving a very high ratio of cache hits at every level. Fortunately, hit ratios of 95% and higher are commonly achieved.

Understanding Prefetching

One way to disguise the latency of memory access is to overlap this time with other work. This is the reason for the out-of-order execution capability of the R10000 CPU, which can suspend as many as four instructions that are waiting for data, and continue to execute following instructions that have their data. This is sufficient to hide most of the latencyv of the L1 cache and often that of the L2 cache as well. Something more is needed to overlap main memory latency with work: data *prefetch*.

Prefetch instructions are part of the MIPS IV instruction set architecture (see "MIPS IV Instruction Set Architecture" on page 21). They allow data to be moved into the cache before their use. If the program requests the data far enough in advance, main-memory latency can be completely hidden. A prefetch instruction is similar to a load instruction: an address is specified and the datum at that address is accessed. The difference is that no register is specified to receive the datum. The CPU checks availability of the datum and, if there is a cache miss, initiates a fetch, so the data ends up in the cache.

There is no syntax in standard languages such as C and Fortran to request a prefetch, so you normally rely on the compiler to automatically insert prefetch instructions. Compiler directives are also provided to allow you to specify prefetching manually.

Principles of Good Cache Use

The operation of the caches immediately leads to some straightforward principles. Two of the most obvious are:

- A program ought to make use of every word of every cache line that it touches.

  Clearly, if a program does not make use of every word in a cache line, the time spent loading the unused parts of the line is wasted.

- A program should use a cache line intensively and then not return to it later.

  When a program visits a cache line a second time after a delay, the cache line may have been displaced by other data, so the program is delayed twice waiting for the same data.

Using Stride-One Access

These two principles together imply that every loop should use a memory *stride* of 1; that is, a loop over an array should access array elements from adjacent memory addresses. When the loop "walks" through memory by consecutive words, it uses every word of every cache line in sequence, and does not return to any cache line after finishing it.

Consider the loop nest in Example 6-1.

Example 6-1 Simple Loop Nest with Poor Cache Use

```
do i = 1, n
   do j = 1, n
      a(i,j) = b(i,j)
   enddo
enddo
```

In Example 6-1, the array *b* is copied to the array *a* one row at a time. Because Fortran stores matrices in column-major order—that is, consecutive elements of a column are in consecutive memory locations—this loop accesses memory with a stride of *n*. Each element loaded or stored is *n* elements away from the last one. As a result, a new cache line must be brought in for each element of *a* and of *b*. If the matrix is small enough, all of it will shortly be brought into cache, displacing other data that was there. However, if the array is too large (the L1 cache is only 32 KB in size), cache lines holding elements at the beginning of a row may be displaced from the cache before the next row is begun. This causes a cache line to be loaded multiple times, once for each row it spans.

Simply reversing the order of the loops, as in Example 6-2, changes the loop to stride-one access.

Example 6-2 Reversing Loop Nest to Achieve Stride-One Access

```
do j = 1, n
   do i = 1, n
      a(i,j) = b(i,j)
   enddo
enddo
```

Using this ordering, the data are copied one column at a time. In Fortran, this means copying adjacent memory locations in sequence. Thus, all data in the same cache line are used, and each line needs to be loaded only once.

Grouping Data Used at the Same Time

Consider the Fortran code segment in Example 6-3, which performs a calculation involving three vectors, *x*, *y*, and *z*.

Example 6-3 Loop Using Three Vectors

```
d = 0.0
do i = 1, n
   j = ind(i)
   d = d + sqrt(x(j)*x(j) + y(j)*y(j) + z(j)*z(j))
enddo
```

Access to the elements of the vectors is not sequential, but indirect through an index array, *ind*. It is not likely that the accesses are stride-one in any vector. Thus, each iteration of the loop may well cause three new cache lines to be loaded, one from each of the vectors.

It appears that *x(j)*, *y(j)*, and *z(j)*, for one value of *j*, are a triple that is always processed together. Consider Example 6-4, in which the three vectors have been grouped together as the three rows of a two-dimensional array, *r*. Each column of the array contains one of the triples *x*, *y*, *z*.

Example 6-4 Three Vectors Combined in an Array

```
d = 0.0
do i = 1, n
   j = ind(i)
   d = d + sqrt(r(1,j)*r(1,j) + r(2,j)*r(2,j) + r(3,j)*r(3,j))
enddo
```

Because *r(1,j)*, *r(2,j)*, and *r(3,j)*—the three rows of column *j*—are contiguous in memory, it is likely that all three values used in one loop iteration will fall in the same cache line. This reduces traffic to the cache by as much as a factor of three. There could be a cache line fetched for each iteration of the loop, but only one, not three.

Suppose the loop in Example 6-3 had not used indirect indexing, but instead progressed through the vectors sequentially, as in this example:

```
d = d + sqrt(x(i)*x(i) + y(i)*y(i) + z(i)*z(i))
```

Then the accesses would have been stride-1, and there would be no advantage to grouping the vectors—unless they were aligned in memory so that *x(i)*, *y(i)*, and *z(i)* all mapped to the same location in the cache. Such unfortunate alignments can happen, and cause what is called cache thrashing.

Understanding Cache Thrashing

Consider the code fragment in Example 6-5, where three large vectors are combined into a fourth.

Example 6-5 Fortran Code Likely to Cause Thrashing

```
parameter (max = 1024*1024)
dimension a(max), b(max), c(max), d(max)
do i = 1, max
   a(i) = b(i) + c(i)*d(i)
enddo
```

The four vectors are declared one after the other, so they are allocated contiguously in memory. Each vector is 4 MB in size (1024 × 1024 times the size of a default real, 4 bytes). Thus the low 22 bits of the addresses of elements *a(i)*, *b(i)*, *c(i)*, and *d(i)* are the same. The vectors map to the same locations in the associative caches.

To perform the calculation in iteration *i* of this loop, *a(i)*, *b(i)*, *c(i)*, and *d(i)* must be resident in the primary cache. *c(i)* and *d(i)* are needed first to carry out the multiplication. They map to the same location in the cache, but both can be resident simultaneously because the cache is two-way set associative—it can tolerate two lines using the same address bits. To carry out the addition, *b(i)* needs to be in the cache. It maps to the same cache location, so the line containing *c(i)* is displaced to make room for it (here we assume that *c(i)* was accessed least recently).To store the result in *a(i)*, the cache line holding *d(i)* must be displaced.

Now the loop proceeds to iteration *i*+1. The line containing *c(i+1)* must be loaded anew into the cache, because one of the following things is true:

- *c(i+1)* is in a different line than *c(i)* and so its line must be loaded for the first time.
- *c(i+1)* is in the same line as *c(i)* but was displaced from the cache during the previous iteration.

Similarly, the cache line holding *d(i+1)* must also be reloaded. In fact, *every* reference to a vector element results in a cache miss, because only two of the four values needed during each iteration can reside in the cache simultaneously. Even though the accesses are stride-one, there is no cache line reuse.

This behavior is known as *cache thrashing*, and it results in very poor performance, essentially reducing the program to uncached use of memory. The cause of the thrashing is the unfortunate alignment of the vectors: they all map to the same cache location. In Example 6-5 the alignment is especially bad because *max* is large enough to cause thrashing in both primary and secondary caches.

Using Array Padding to Prevent Thrashing

There are two ways to repair cache thrashing:

1. Redimension the vectors so that their size is not a power of two. A new size that spaces the vectors out in memory so that *a(1)*, *b(1)*, *c(1)* and *d(1)* all map to different locations in the cache is ideal. For example, `max = 1024 × 1024 + 32` would offset the beginning of each vector 32 elements, or 128 bytes. This is the size of an L2 cache line, so each vector begins at a different cache address. All four values may now reside in the cache simultaneously, and complete cache line reuse is possible.
2. Introduce padding variables between the vectors in order to space out their beginning addresses. Ideally, each padding variable should be at least the size of a full cache line. Thus if *max* is kept the same, the following declaration eliminates cache thrashing:

```
      dimension a(max), pad1(32), b(max), pad2(32),
     &          c(max), pad3(32), d(max)
```

3. For multidimensional arrays, it is sufficient to make the leading dimension an odd number, as in the following:

```
dimension a(1024+1,1024)
```

   For arrays with smaller dimensions, it is necessary to change two or more dimensions, as in the following:

```
dimension a(64+1,64+1,64)
```

Eliminating cache thrashing makes the loop at least 100 times faster.

Identifying Cache Problems with Perfex and SpeedShop

The profiling tools discussed in Chapter 4, "Profiling and Analyzing Program Behavior," are used to determine if there is a cache problem. To see how they work, reexamine the profiles taken from *adi2.f* (Example 4-2 on page 55 and Example 4-6 on page 67).

You can review the code of this program in Example C-1 on page 288. It processes a three-dimensional cubical array. The three procedures **xsweep**, **ysweep**, and **zsweep** perform operations along pencils in each of the three dimensions, as indicated in Figure 6-1.

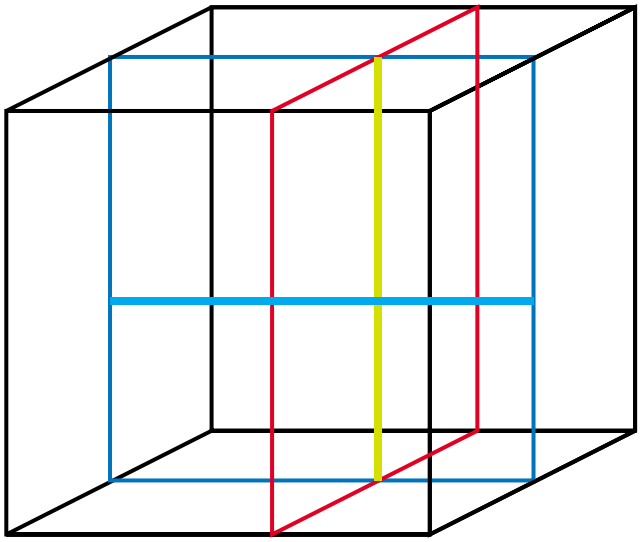


Figure 6-1 Processing Directions in adi2.f

The profiles using PC sampling (Example 4-6 on page 67) presented times for these procedures indicating that the vast majority of time is spent in the routine **zsweep**:

```
samples   time(%)      cum time(%)      procedure (dso:file)
   6688   6.7s( 78.0)  6.7s( 78.0)         zsweep (adi2:adi2.f)
    671  0.67s(  7.8)  7.4s( 85.8)         xsweep (adi2:adi2.f)
    662  0.66s(  7.7)    8s( 93.6)         ysweep (adi2:adi2.f)
```

However, the ideal time profile (Example 4-9 on page 69) presented quite a different picture, showing identical instruction counts in each routine.

PC sampling, which samples of the program counter position as the program runs, obtains an accurate measure of elapsed time associated with memory accesses. Ideal time counts the instructions and cannot tell the difference between a cache hit and a miss.

The difference between the *-pcsamp* and *-ideal* profiles indicates that the instructions in **zsweep** occupied much more elapsed time than similar instructions in the other routines. The explanation is that **zsweep** suffers from some serious cache problems, but it provides no indication of the level of memory—L1 cache, L2 cache, or TLB—that is the principal source of the trouble. This can be determined using the R10000 event counters.

The *perfex -a -y* output for this program appears in Example 4-3 on page 56 including the lines shown in Example 6-6 (only the "Typical" time column is retained).

Example 6-6 Perfex Data for adi2.f

```
16 Cycles.....................................................  1639802080     8.366337
26 Secondary data cache misses................................     7736432     2.920580
23 TLB misses.................................................     5693808     1.978017
 7 Quadwords written back from scache.........................    61712384     1.973562
25 Primary data cache misses..................................    16368384     0.752445
22 Quadwords written back from primary data cache.............    32385280     0.636139
```

The program spent about 3 seconds of its 8-second run in secondary cache misses, and nearly 2 seconds in TLB misses. These results make it likely that both secondary cache and TLB misses are hindering the performance of **zsweep**. To understand why, it is necessary look in more detail at what happens when this routine is called (read the program in Example C-1 on page 288).

If you know that a program is being slowed by memory access problems, but you are not sure which parts of the program are causing the problem, you could perform *ssrun* experiments of types *-dc_hwc*, *-sc_hwc*, and *-tlb_hwc* (see Table 4-2 on page 64). These profiles should highlight exactly the parts of the program that are most delayed. In the current example, we know from the profile in Example 4-6 that the problem is in routine **zsweep**.

Diagnosing and Eliminating Cache Thrashing

zsweep performs calculations along a pencil in the z-dimension. This is the index of the a array that varies most slowly in memory. The declaration for *data* is as follows:

```
parameter (ldx = 128, ldy = 128, ldz = 128)
real*4 data(ldx,ldy,ldz)
```

There are 128 × 128 = 16384 array elements, or 64KB, between successive elements in the z-dimension. Thus, each element along the pencil maps to the same location in the primary cache. In addition, every 32nd element maps to the same place in a 4 MB secondary cache (every eighth element, for a 1 MB cache). The pencil is 128 elements long, so the data in the pencil map to four different secondary cache lines (16, for a 1 MB L2 cache). Since the caches are only two-way set associative, there will be cache thrashing.

As we saw (“Understanding Cache Thrashing” on page 140), cache thrashing is fixed by introducing padding. All the data are in a single array, so you can introduce padding by increasing the array bounds. Changing the array declaration to the following provides sufficient padding:

```
parameter (ldx = 129, ldy = 129, ldz = 129)
real*4 data(ldx,ldy,ldz)
```

With these dimensions, array elements along a z-pencil all map to different secondary cache lines. (For a system with a 4MB cache, two elements, *a(i,j,k1)* and *a(i,j,k2)*, map to the same secondary cache line when ((k2 - k1) × 129 × 129 × 4 mod (4MB ÷ (2 associative sets))) ÷ 128 = 0.)

Running this modification of the program produces the counts shown in Example 6-7.

Example 6-7 Perfex Data for adi5.f

```
16 Cycles.....................................................    453456768     1.813827
26 Secondary data cache misses................................      1623296     0.490235
23 TLB misses.................................................      6262112     1.705549
7 Quadwords written back from scache..........................     12049152     0.308458
25 Primary data cache misses..................................     11676544     0.420823
22 Quadwords written back from primary data cache.............     16002160     0.246433
```

The secondary cache misses have been reduced by more than a factor of six. The primary cache misses have also been greatly reduced. Now, TLB misses represent the dominant memory cost, at 1.7 seconds of the program’s 1.8-second run time.

Diagnosing and Eliminating TLB Thrashing

The reason TLB misses consume such a large percentage of the time is thrashing of that cache. Each TLB entry stores the real-memory address of an even-odd pair of virtual memory pages. Each page is 16 KB in size (by default), and adjacent elements in a z-pencil are separated by 129 × 129 × 4 = 65564 bytes, so every z-element falls in a separate TLB entry. However, there are 128 elements per pencil and only 64 TLB entries, so the TLB must be completely reloaded (twice) for each sweep in the z-direction; there is no TLB cache reuse. (Note that this does not mean the data is not present in memory. There are no page faults happening. However, the CPU can address only a limited number of pages at one time. What consumes the time is the frequent traps to the kernel to reload the TLB registers for different virtual pages.)

There are a couple of ways to fix this problem: cache blocking and copying. Cache blocking is an operation that the compiler loop-nest optimizer can perform automatically (see "Controlling Cache Blocking" on page 167). Copying is a trick that the optimizer does not perform, but that is not difficult to do manually. Cache blocking can be used in cases where the work along z-pencils can be carried out independently on subsets of the pencil. But if the calculations along a pencil require a more complicated intermixing of data, or are performed in a library routine, copying is the only general way to solve this problem.

Using Copying to Circumvent TLB Thrashing

The TLB thrashing results from having to work on data that are spread over too many virtual pages. This is not a problem for the x- and y-dimensions because these data are close enough together to allow many sweeps to reuse the same TLB entries. What is needed is a way to achieve the same kind of reuse for the z-dimension. This can be accomplished by a three-step algorithm:

- Copy each z-pencil to a scratch array that is small enough to avoid TLB thrashing.
- Carry out the z-sweep on the scratch array.
- Copy the results back to the *data* array.

How big should this scratch array be? To avoid thrashing, it needs to be smaller than the number of pages that the TLB can map simultaneously; this is approximately 58×2 pages (IRIX reserves a few of the 64 TLB entries for its own use), or 1,856 KB. But it shouldn't be too small, because the goal is to get a lot of reuse from the page mappings while they are resident in the TLB.

The calculations in both the x- and y-dimensions achieve good performance. This indicates that an array the size of an xz-plane should work well. An xz-plane is used rather than a yz-plane to maximize cache reuse in the x-dimension. If an entire plane requires more scratch space than is practical, a smaller array can be used, but to get good secondary cache reuse, a subset of an xz-plane whose x-size is a multiple of a cache line should be used.

A version of the program modified to copy entire xz-planes to a scratch array before z-sweeps are performed (see Example C-3 on page 291) produces the counts shown in Example 6-8.

Example 6-8 Perfex Data for adi53.f

```
16 Cycles.....................................................    354393968     1.417576
25 Primary data cache misses..................................     10999808     0.396433
22 Quadwords written back from primary data cache.............     16251344     0.250271
26 Secondary data cache misses................................      1126256     0.340129
 7 Quadwords written back from scache.........................     10343456     0.264792
23 TLB misses.................................................        79568     0.021671
```

Although the frequent copying of z-data increases the primary and secondary cache misses somewhat (compare Example 6-8 to Example 6-7), the TLB misses all but disappear, and overall performance is significantly improved. Thus the overhead of copying is a good trade-off.

In cases where it is possible to use cache blocking instead of copying, the TLB miss problem can be solved without increasing the primary and secondary cache misses and without expending time to do the copying, so it will provide even better performance. Use copying when cache blocking is not an option.

Using Larger Page Sizes to Reduce TLB Misses

An alternate technique, one that does not require code modification, is to tell the operating system to use larger page sizes. Each TLB entry represents two adjacent pages; when pages are larger, each entry represents more data. Of course, if you then attempt to work on larger problems, the TLB misses can return, so this technique is more of an expedient than a general solution. Nevertheless, it's worth looking into.

The page size can be changed in two ways: The utility *dplace* can reset the page size for any program. Its use is very simple (see the dplace(1) reference page):

```
% dplace -data_pagesize 64k -stack_pagesize 64k program arguments...
```

With this command, the data and stack page sizes are set to 64 KB during the execution of the specified *program*. TLB misses typically come as a result of calculations on the data in the program, so setting the page size for the program text (the instructions)—which would be done with the option *-text_pagesize 64k*—is generally not needed.

For parallel programs that use the MP library, a second method exists. All you need do is to set some environment variables. For example, the following three lines set the page size for data, text and stack to 64 KB:

```
% setenv PAGESIZE_DATA  64
% setenv PAGESIZE_STACK 64
% setenv PAGESIZE_TEXT  64
```

These environment variables are acted on by the runtime multiprogramming library (*libmp*) and are effective only when the program is linked *-lmp*.

There are some restrictions, on what page sizes can be used. First of all, the only valid page sizes are 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, and 16 MB. Second, the system administrator must specify what percentage of memory is allocated to the various possible page sizes. Thus, if the system has not been set up to allow 1 MB pages, requesting this page size will be of no use.

The administrator can change the page size allocation percentages at run-time, for the entire system, using the *systune* command to change the system variables percent_totalmem_*_pages (see the *System Configuration and Operation* manual mentioned under "Related Manuals" on page xxix, Chapter 10).

Using Other Cache Techniques

Through the use of *perfex*, PC sampling, and ideal time profiling, cache problems can easily be identified. Two techniques for dealing with them have already been covered ("Understanding Cache Thrashing" on page 140 and "Using Copying to Circumvent TLB Thrashing" on page 146). Other useful methods for improving cache performance are loop fusion, cache blocking, and data transposition.

Understanding Loop Fusion

Consider the pair of loops in Example 6-9.

Example 6-9 Sequence of DAXPY and Dot-Product on a Single Vector

```
dimension a(N), b(N)
do i = 1, N
   a(i) = a(i) + alpha*b(i)
enddo
dot = 0.0
do i = 1, N
   dot = dot + a(i)*a(i)
enddo
```

These two loops carry out two vector operations. The first is the DAXPY operation we have seen before (Example 5-3 on page 99); the second is a dot-product. If *n* is large enough that the vectors don't fit in cache, the code will stream the vector *b* from memory

through the cache once and the vector *a* twice: once to perform the DAXPY operation, and a second time to calculate the product. This violates the second general principle of cache management ("Principles of Good Cache Use" on page 138), that a program should use each cache line once and not revisit it.

The code in Example 6-10 performs the same work as Example 6-9 but in one loop.

Example 6-10 DAXPY and Dot-Product Loops Fused

```
dimension a(N), b(N)
dot = 0.0
do i = 1, N
   a(i) = a(i) + alpha*b(i)
   dot  = dot + a(i)*a(i)
enddo
```

Now the vector *a* only needs to be streamed through the cache once because its contribution to the dot-product can be calculated while the cache line holding *a(i)* is still resident. This version takes approximately one-third less time than the original version.

Combining two loops into one is called *loop fusion*. One potential drawback to fusing loops, however, is that it makes the body of the fused loop larger. If the loop body gets too large, it can impede software pipelining. The Loop-Nest Optimizer tries to balance the conflicting goals of better cache behavior with efficient software pipelining.

Understanding Cache Blocking

Another standard technique used to improve cache performance is *cache blocking*, or cache tiling. Here, data structures that are too big to fit in the cache are broken up into smaller pieces that will fit in the cache. As an example, consider the matrix multiplication code in Example 6-11.

Example 6-11 Matrix Multiplication Loop

```
do j = 1, n
   do i = 1, m
      do k = 1, l
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
   enddo
enddo
```

If arrays *a*, *b*, and *c* are small enough that all fit in cache, performance is great. But if they are too big, performance drops substantially, as can be seen from the results in Table 6-1

Table 6-1 Cache Effect on Performance of Matrix Multiply

m	n	p	Seconds	MFLOPS
30	30	30	0.000162	333.9
200	200	200	0.056613	282.6
1000	1000	1000	25.43118	78.6

(The results in Table 6-1 were obtained with LNO turned off because the LNO is designed to fix these problems!) The reason for the drop in performance is clear if you use *perfex* on the 1000 × 1000 case:

```
 0 Cycles.....................................  5460189286  27.858109
23 TLB misses.................................    35017210  12.164907
25 Primary data cache misses..................   200511309   9.217382
26 Secondary data cache misses................    24215965   9.141769
```

What causes this heavy memory traffic? To answer this question, we need to count the number of times each matrix element is touched. From reading the original code in Example 6-11, we can see that the memory access pattern shown schematically in Figure 6-2.

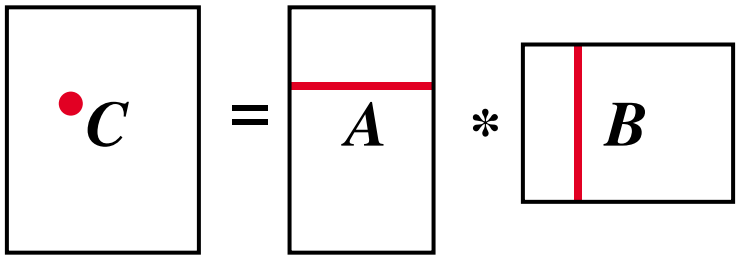


Figure 6-2 Memory Use in Matrix Multiply

Calculating one element of *c* requires reading an entire row of *a* and an entire column from *b*. Calculating an entire column of *c* requires reading all the rows of *a*, so *a* is swept *n* times, once for each column of *c* (or, equivalently, each column of *c*). Similarly, every column of *b* must be reread for each element of *c*, so *b* is swept *m* times.

If *a* and *b* don't fit in the cache, the earlier rows and columns are likely to be displaced by the later ones, so there is likely to be little reuse. As a result, these arrays will require streaming data in from main memory, *n* times for *a* and *m* times for *b*.

Because the operations on array cells in Example 6-11 are not order-dependent, this problem with cache misses can be fixed by treating the matrices in sub-blocks as shown in Figure 6-3. In blocking, a block of *c* is calculated by taking the dot-product of a block-row of *a* with a block-column of *b*. The dot-product consists of a series of sub-matrix multiplies. When three blocks, one from each matrix, all fit in cache simultaneously, the elements of those blocks need to be read in from memory only once for each sub-matrix-multiply.

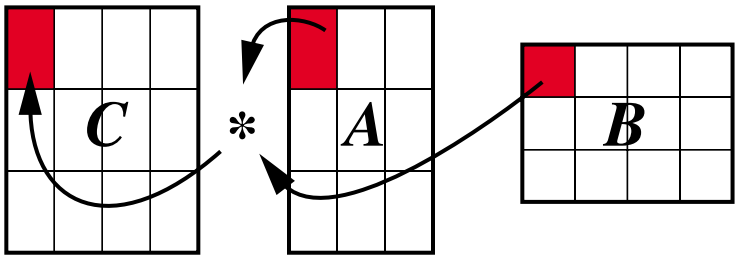


Figure 6-3 Cache Blocking of Matrix Multiplication

The bigger the blocks are, the greater the reduction in memory traffic—but only up to a point. Three blocks must be able to fit into cache at the same time, so there is an upper bound on how big the blocks can be. For example, three 200x200 double-precision blocks take up almost 1 MB, so this is about as big as the blocks can be on a 1 MB cache system. (Most SN0 systems now have 4 MB caches or larger. The *hinv* command reports the cache size.) Furthermore, the addresses of the blocks must not conflict with themselves or one another; that is, elements of different blocks must not map to the same cache lines (see "Understanding Cache Thrashing" on page 140). Conflicts can be reduced or eliminated by proper padding of the leading dimension of the matrices, by careful alignment of the matrices in memory with respect to one another, and, if necessary, by further limiting of the block size.

One other factor affects what size blocks can be used: the TLB cache. The larger we make the blocks, the greater the number of TLB entries that will be required to map their data. If the blocks are too big, the TLB cache will thrash (see "Diagnosing and Eliminating TLB Thrashing" on page 145). For small matrices, in which a pair of pages spans multiple columns of a submatrix, this is not much of a restriction. But for very large matrices, this means that the width of a block will be limited by the number of available TLB entries.

(The submatrix multiplies can be carried out so that data accesses go down the columns of two of the three submatrixes and across the rows of only one submatrix. Thus, the number of TLB entries only limits the width of one of the submatrixes.) This is just one more detail that must be factored into how the block sizes are chosen.

Return now to the 1000 × 1000 matrix multiply example. Each column is 8,000 bytes in size. Thus (1048576 ÷ 2) ÷ 8000 = 65 columns will fit in one associative set of a 1 MB secondary cache, and 130 will fill both sets. In addition, each TLB entry maps four columns, so the block size is limited to about 225 before TLB thrashing starts to become a problem. The performance results for three different block sizes is shown in Table 6-2.

Table 6-2 Cache Blocking Performance of Large Matrix Multiply

m	n	p	block order	mbs	nbs	pbs	Seconds	MFLOPS
1000	1000	1000	ijk	65	65	65	6.837563	292.5
1000	1000	1000	ijk	130	130	130	7.144015	280.0
1000	1000	1000	ijk	195	195	195	7.323006	273.1

For block sizes that limit cache conflicts, the performance of the blocked algorithm on this larger problem matches the performance of the unblocked code on a size that fits entirely in the secondary cache (the 200 × 200 case in Table 6-1). The *perfex* counts for the block size of 65 are as follows:

```
 0 Cycles..................................... 1406325962   7.175132
21 Graduated floating point instructions...... 1017594402   5.191808
25 Primary data cache misses.................   82420215   3.788806
26 Secondary data cache misses...............    1082906   0.408808
23 TLB misses................................     208588   0.072463
```

The secondary cache misses have been greatly reduced and the TLB misses are now almost nonexistent. From the times *perfex* estimates, the primary cache misses look to be a potential problem (as much as 3.8 seconds of the run time). However, the *perfex* estimate assumes no overlap of primary cache misses with other work. The R10000 CPU manages to achieve a lot of overlap on these misses, so this "typical" time is an overestimate. This can be confirmed by looking at the times of the other events and comparing them with the total time. The inner loop of matrix multiply generates one madd instruction per cycle, and *perfex* estimates the time based on this same ratio, so the floating point time should be pretty accurate. On the basis of this number, the primary cache misses can account for, at most, two seconds of time. (This points out that, when estimating primary cache misses, you have to make some attempt to account for overlap, either by making an educated guess or by letting the other counts lead you to reasonable bounds.)

Square blocks are used for convenience in the example above. But there is some overhead to starting up and winding down software pipelines, so rectangular blocks, in which the longest dimension corresponds to the inner loop, generally perform the best.

Cache blocking is one of the optimizations the Loop-Nest Optimizer performs. It chooses block sizes that are as large as it can fit into the cache without conflicts. In addition, it uses rectangular blocks to reduce software pipeline overhead and maximize data use from each cache line. If we simply compile the matrix multiply kernel with *-O3* or *-Ofast*, which enable LNO, the unmodified code of Example 6-11 with 1000×1000 arrays achieves the same speed as the best run shown in Table 6-2.

But the LNO's automatic cache blocking will not solve all possible cache problems automatically. Some require manual intervention.

Understanding Transpositions

Fast Fourier transforms (FFT) algorithms present challenges on a cache-based machine because of their data access patterns, and because they usually involve arrays that are a power of two in size.

In a radix-2 FFT implementation, there are $\log(n)$ computational stages. In each stage, pairs of values are combined. In the first stage, these data are $n/2$ elements apart; in the second stage, data that are $n/4$ elements apart are combined; and so on. Figure 6-4 is a diagram showing the three stages of an 8-point transform.

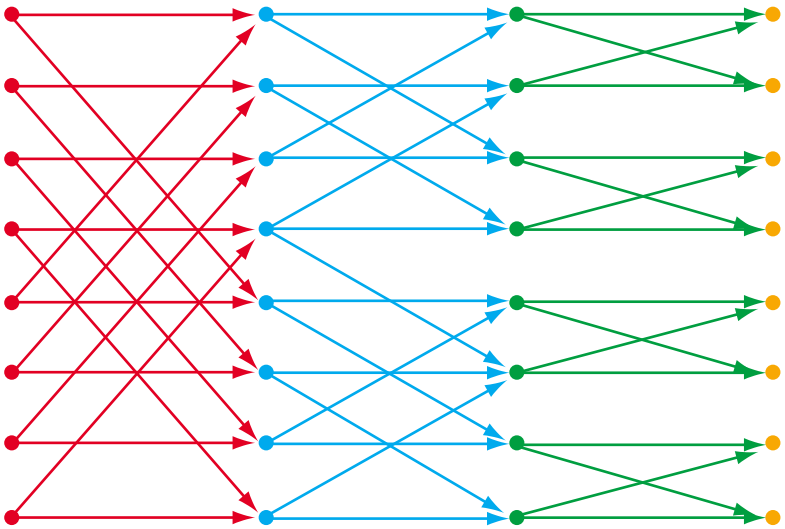


Figure 6-4 Schematic of Data Motion in Radix-2 Fast Fourier Transform

Both the size of the cache and the separation between combined data elements is a power of 2, so this data access pattern is likely to cause cache conflicts when *n* is large enough that the data do not fit in the primary cache. The R10000 caches are two-way set associative, and the two-way conflicts of the radix-2 algorithm are resolved without thrashing. However, a radix-2 implementation is not particularly efficient. A radix-4 implementation, in which four elements are combined at each stage, is much better: it uses half the number of stages—which means it makes half the memory accesses—and it requires fewer arithmetic operations. But cache address conflicts among four data elements are likely cause thrashing.

The solution is to view the data as a two-dimensional, rather than a one-dimensional, array. Viewed this way, half the stages of the algorithm become FFTs across the rows of the array, while the remainder are FFTs down the columns. The column FFTs are cache friendly (assuming Fortran storage order), so nothing special needs to be done for them.

The row FFTs can suffer from cache thrashing. The easiest way to eliminate the thrashing is through the use of copying (see "Using Copying to Circumvent TLB Thrashing" on page 146).

An alternative that results in slightly better performance is to transpose the two-dimensional array, so that cache-thrashing row FFTs become cache-friendly column FFTs.This method is described at a high level here. However, library algorithms for FFT are already tuned to implement this method, so there is rarely a need to reimplement it.

To carry out the transpose, partition the two-dimensional array into square blocks. If the array itself cannot be a square, it can have twice as many rows as columns, or vice versa. In this case, it can be divided into two squares; these two squares can be transposed independently.

The number of columns in a block is chosen to be that number that will fit in the cache without conflict (two-way conflicts are all right because of the two-way set associativity). The square array is then transposed as follows:

- Off-diagonal blocks in the first block-column and block-row are transposed and exchanged with one another. At this point, the entire first column will be in the cache.
- Before proceeding with the rest of the transpose, perform the FFTs down the first block-column on the in-cache data.
- Once these have been completed, exchange the next block-column and block-row. These are one block shorter than in the previous exchange; as a result, once the exchange is completed, not all of the second block-column is guaranteed to be in the cache. But most of it is, so FFTs are carried on this second block-column.
- Continue exchanges and FFTs until all the row FFTs have been completed.

Now we are ready for the column FFTs, but there is a problem. The columns are now rows, and so the FFTs will cause conflicts. This is cured just as above: transpose the array, but to take greatest advantage of the data currently in the cache, proceed from right to left rather than from left to right.

When this transpose and the column FFTs have been completed, the work is almost done. A side effect of treating the one-dimensional FFT as a two-dimensional FFT is that the data end up transposed. Once again this is no problem; they can simply be transposed a third time.

Despite the three transpositions, this algorithm is significantly faster than allowing the row FFTs to thrash the cache, for data that don’t fit in L1 But what about data that do not even fit in the secondary cache? Treat these larger arrays in blocks just as above, but if the FFTs down the columns of the transposed array don’t fit in the primary cache, use another level of blocking and transposing for them.

Any well-tuned library routine for calculating FFT requires three separate algorithms: a textbook radix-4 algorithm for the small data sets that fit in L1; a single-blocked algorithm for the intermediate sizes; and a double-blocked algorithm for large data sets. The FFTs in CHALLENGE complib and SCSL are designed in just this way (see "Exploiting Existing Tuned Code" on page 49).

Summary

Each level of the memory hierarchy is at least an order of magnitude faster than the level below it. The basic guidelines of good cache management follow from this: fetch data from memory only once, and, when data has been fetched, use every word of it before moving on. Perfectly innocent loops can violate these guidelines when they are applied to large arrays, with the result that the program slows down drastically.

In order to diagnose memory access problems, look for cases where a profile based on real-time sampling differs from an ideal-time profile, or use *ssrun* experiments to isolate the delayed routines. Then use *perfex* counts to find out which levels of the memory hierarchy are the problem. Once the problem is clear, it can usually be fixed by inverting the order of loops so as to proceed through memory with stride one, or by copying scattered data into a compact buffer, or by transposing to make rows into columns. Or, as the next chapter details, the compiler can often take care of the problem automatically.

Chapter 7

Using Loop Nest Optimization

When you compile with the highest optimization levels (*-O3* or *-Ofast*), the compiler applies the Loop-Nest Optimizer (LNO). The LNO is capable of solving many cache-use problems automatically. When you know what it can do, you can use it to implement more sophisticated cache optimizations. This chapter surveys the features of the LNO in the following topics:

Understanding Loop Nest Optimizations

The LNO attempts to improve cache and instruction scheduling by performing high-level transformations on the source code of loops. Some of the changes the LNO can make have been presented as manual techniques (for example, "Diagnosing and Eliminating Cache Thrashing" on page 144 and "Understanding Loop Fusion" on page 148). Often the LNO applies such techniques automatically. Many LNO optimizations have effects beyond improved cache behavior. For example, software pipelining is more effective following some of the LNO transformations.

Much of the time you can accept the default actions of the LNO and get good results, but not all loop nest optimizations improve program performance all the time, so there are instances in which you can improve program performance by turning off some or all of the LNO options.

Requesting LNO

The LNO is turned on automatically when you use the highest level of optimization, *-O3* or *-Ofast*. You send specific options to the LNO phase using the -LNO: compiler option group. The numerous sub-options are documented in the LNO(5) reference page. To restrict the LNO from transforming loops while performing other optimizations, use *-O3 -LNO:opt=0*.

You can invoke specific LNO features directly in the source code. In C, you use pragma statements; in Fortran, directives. The pragmas and directives are covered in the LNO(5) reference page. Source directives take precedence over the command-line options, but they can be disabled with the flag *-LNO:ignore_pragmas*.

When the compiler cannot determine the run-time size of data structures, it assumes they will not fit in the cache, and it carries out all the cache optimizations it is capable of. As a result, unnecessary optimizations can be performed on programs that are already cache-friendly. This typically occurs when the program uses an array whose size is not declared in the source. When this seems to be a problem, disable LNO transformations either for the entire module, or for a specific loop using a pragma or directive.

Reading the Transformation File

You can find out what transformations are performed by LNO, IPA, and other components of the compiler in a transformation file. This is a C or Fortran source file emitted by the compiler that shows what the program looks like after the optimizations have been applied.

You cause a Fortran transformation file to be emitted by using the *-flist* or *-FLIST:=ON* flags; a C file will be emitted if the *-clist* or *-CLIST:=ON* flags are used. The transformation file is given the same name as the source file being compiled, but *.w2f* or *.w2c* is inserted before the *.f* or *.c* suffix; for example the transformation file for *test.f* is *test.w2f.f*. The compiler issues a diagnostic message indicating that the transformation file has been written and what its name is.

When IPA is enabled (“Requesting IPA” on page 126), the transformation file is not written until the link step. Without IPA, the transformation file is written when the module is compiled.

To be as readable as possible, the transformation file uses as many of the variable names from the original source as it can. However, Fortran files are written using a free format in which lines begin with a tab character and do not adhere to the 72-character limit. The MIPSpro compilers compile this form correctly, but if you wish the file to be written using the standard 72-character Fortran layout, use the *-FLIST:ansi_format* flag.

The LNO's transformations complicate the code of loops. It is educational to see what it has done, and you can judge whether a particular loop justified the changes that are made to it.

In addition to the transformation file, recall that the compiler writes software pipelining report cards into the assembler file it emits when the *-S* option is used (see "Reading Software Pipelining Messages" on page 105).

Using Outer Loop Unrolling

One of the noncache optimizations that the LNO performs is outer loop unrolling (sometimes called register blocking, because it tends to get a small block of array values into registers for multiple operations). Consider the subroutine in Example 7-1, which implements a matrix multiplication.

Example 7-1 Matrix Multiplication Subroutine

```
subroutine mm(a,lda,b,ldb,c,ldc,m,l,n)
integer lda, ldb, ldc, m, l, n
real*8 a(lda,l), b(ldb,n), c(ldc,n)
do j = 1, n
   do k = 1, l
      do i = 1, n
         c(i,j) = c(i,j) + a(i,k)*b(k,j)
      enddo
   enddo
enddo
return
end
```

The SWP report card for this loop (assuming we compile with loop nest transformations turned off) is as shown in Example 7-2.

Example 7-2 SWP Report Card for Matrix Multiplication

```
#<swps> Pipelined loop line 7 steady state
#<swps>
#<swps>     2 unrollings before pipelining
#<swps>     6 cycles per 2 iterations
#<swps>     4 flops        ( 33% of peak) (madds count as 2)
#<swps>     2 flops        ( 16% of peak) (madds count as 1)
#<swps>     2 madds        ( 33% of peak)
#<swps>     6 mem refs     (100% of peak)
#<swps>     2 integer ops  ( 16% of peak)
#<swps>    10 instructions ( 41% of peak)
#<swps>     1 short trip threshold
#<swps>     6 ireg registers used.
#<swps>     8 fgr registers used.
```

Example 7-2 is just like the report card for the DAXPY operation (Example 5-7 on page 106). It says that, in the steady state, this loop will achieve 33% of the peak floating-point rate, and the reason for this less-than-peak performance is that the loop limited by is loads and stores to memory. But is it really?

Software pipelining is applied to only a single loop. In the case of a loop nest, as in Example 7-1, the innermost loop is software pipelined. The inner loop is processed as if it were independent of the loops surrounding it. But this is not the case in general, and it is certainly not the case for matrix multiply.

Consider the modified loop nest in Example 7-3.

Example 7-3 Matrix Multiplication Unrolled on Outer Loop

```
subroutine mm1(a,lda,b,ldb,c,ldc,m,l,n)
integer lda, ldb, ldc, m, l, n
real*8 a(lda,l), b(ldb,n), c(ldc,n)
do j = 1, n, nb
   do k = 1, l
      do i = 1, n
         c(i,j+0) = c(i,j+0) + a(i,k)*b(k,j+0)
         c(i,j+1) = c(i,j+1) + a(i,k)*b(k,j+1)
                             .
                             .
                             .
         c(i,j+nb-1) = c(i,j+nb-1) + a(i,k)*b(k,j+nb-1)
      enddo
   enddo
enddo
return
end
```

Example 7-3 is the same as Example 7-1 except that the outermost loop has been unrolled *nb* times. (To be truly equivalent, the upper limit of the *j*-loop should be changed to *n-nb+1* and an additional loop nest added to take care of leftover iterations when *nb* does not divide *n* evenly. These details have been omitted for ease of presentation.)

The thing to note about Example 7-3 in comparison Example 7-1 is that it processes *nb* different *j*-values in the inner loop, and *a(i,k)* needs to be loaded into a register only once for all these calculations. Because of the order of operations in Example 7-1, the same *a(i,k)* value needs to be loaded once for each of the *c(i,j+0)*, ... *c(i,j+nb-1)* calculations, that is, a total of *nb* times. Thus, the outer loop unrolling has reduced by a factor of *nb* the number of times that elements of matrix *a* need to be loaded.

Now consider the modified loop nest in Example 7-4.

Example 7-4 Matrix Multiplication Unrolled on Middle Loop

```
subroutine mm2(a,lda,b,ldb,c,ldc,m,l,n)
integer lda, ldb, ldc, m, l, n
real*8 a(lda,l), b(ldb,n), c(ldc,n)
do j = 1, n
   do k = 1, l, lb
      do i = 1, n
         c(i,j) = c(i,j) + a(i,k+0)*b(k+0,j)
         c(i,j) = c(i,j) + a(i,k+1)*b(k+1,j)
                         .
                         .
                         .
         c(i,j) = c(i,j) + a(i,k+lb-1)*b(k+lb-1,j)
      enddo
   enddo
enddo
return
end
```

Here, the middle loop has been unrolled *lb* times. Because of the unrolling, *c(i,j)* needs to be loaded and stored only once for all *lb* calculations in the inner loop. Thus, unrolling the middle loop reduces the loads and stores of the c matrix by a factor of *lb*.

If both outer and middle loops are unrolled simultaneously, then both the load and store reductions occur. This code is shown in Example 7-5.

Example 7-5 Matrix Multiplication Unrolled on Outer and Middle Loops

```
do] j = 1, n, nb
         do k = 1, l, lb
            do i = 1, n
               c(i,j) = c(i,j+0) + a(i,k+0)*b(k+0,j+0)
               c(i,j) = c(i,j+0) + a(i,k+1)*b(k+1,j+0)
                                   .
                                   .
                                   .
               c(i,j) = c(i,j+0) + a(i,k+lb-1)*b(k+lb-1,j+0)
               c(i,j) = c(i,j+1) + a(i,k+0)*b(k+0,j+1)
               c(i,j) = c(i,j+1) + a(i,k+1)*b(k+1,j+1)
                                   .
                                   .
                                   .
               c(i,j) = c(i,j+1) + a(i,k+lb-1)*b(k+lb-1,j+1)
                                   .
                                   .
                                   .
               c(i,j) = c(i,j+nb-1) + a(i,k+0)*b(k+0,j+0)
               c(i,j) = c(i,j+nb-1) + a(i,k+1)*b(k+1,j+0)
                                   .
                                   .
                                   .
               c(i,j) = c(i,j+nb-1) + a(i,k+lb-1)*b(k+lb-1,j+nb-1)
            enddo
         enddo
      enddo
```

The total operation count in the inner loop of Example 7-5 is as follows:

madds	lb * nb
c loads	nb
c stores	nb
a loads	lb
b loads	none—the *b* subscripts, *j* and *k,* do not vary in the inner loop; if there are enough machine registers, all needed *b* values are loaded in the middle loop and remain in registers throughout the inner loop.

With no unrolling (*nb* = *lb* = 1), the inner loop is memory-bound. However, the number of madds grows quadratically with the unrolling, while the number of memory operations grows only linearly. By increasing the unrolling, it may be possible to convert the loop from memory-bound to floating point-bound.

Accomplishing this requires that the number of madds be at least as large as the number of memory operations. Although a branch and a pointer increment must also be executed by the CPU, the integer ALUs are responsible for these operations. For the R10000 CPU there are plenty of superscalar slots for them to fit in, so they don't figure in the selection of the unrolling factors. (This is not the case for R8000, because memory operations and madds together can fill up all the superscalar slots.)

The table in Figure 7-1 shows how changing the loop-unrolling counts *lb* and *nb* affect the software pipelining of Example 7-5.

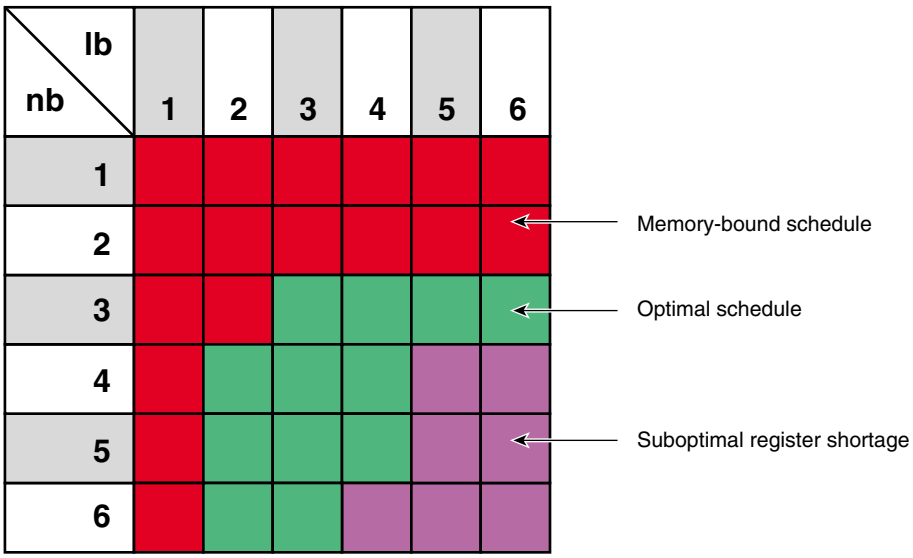


Figure 7-1 Table of Loop-Unrolling Parameters for Matrix Multiply

Several blocking choices produce schedules achieving 100% of floating-point peak. Keep in mind, though, that this reflects the steady-state performance of the inner loop for in-cache data. Some overhead is incurred in filling and draining the software pipeline, and this overhead varies depending on the unrolling values used. In addition, if the arrays don't fit in the cache, the performance will be substantially lower (unless the array blocking optimization is also applied).

Outer loop unrolling is one optimization that the LNO performs; it chooses the proper amount of unrolling for loop nests such as this matrix multiply kernel. For this particular case, if you compile with blocking turned off, the compiler chooses to unroll the *j*-loop by two and the *k*-loop by four, achieving a perfect schedule. (With blocking turned on, the compiler blocks the matrix multiply in a more complicated transformation, so it is more difficult to rate the effectiveness of just the outer loop unrolling.)

Controlling Loop Unrolling

Although the LNO often chooses the optimal amount of unrolling, several flags and a directive are provided for fine-tuning. The three principal flags that control outer loop unrolling are these:

-LNO:outer_unroll=*n*	Tells the compiler to unroll every outer loop for which unrolling is possible by exactly *n*. It either unrolls by *n* or not at all. If you use this flag, you cannot use the next two.
-LNO:outer_unroll_max=*n*	Tells the compiler it may unroll any outer loop by as many as *n*, but no more.
-LNO:outer_unroll_prod_max=*n*	Says that the product of unrolling a given loop nest shall not exceed *n*. The default is 16.

These flags apply to all the loop nests in the file being compiled. To control the unrolling of an individual loop, you may use a directive or pragma. For Fortran, directive is

```
c*$* unroll (n)
```

For C, the corresponding pragma is:

```
#pragma unroll (n)
```

These directives apply to the loop immediately following the directive; when it is not innermost in its nest, outer loop unrolling is performed. The value of *n* must be at least one. If *n* = 1, no unrolling is performed. If *n* = 0, the default unrolling is applied.

When this directive immediately precedes an innermost loop, standard loop unrolling is done. This latter use is not recommended because the software pipeliner unrolls inner loops if it finds that beneficial; generally, the software pipeliner is the best judge of how much inner loop unrolling should be done.

Using Loop Interchange

The importance of stride-1 array accesses was explained under "Using Stride-One Access" on page 138. In C, for example, the initialization loop in Example 7-6 touches memory with a stride of *m*.

Example 7-6 Simple Loop Nest with Poor Cache Use

```
for (i=0; i<n; i++) {
   for (j=0; j<m; j++) {
      a[j][i] = 0.0;
   }
}
```

Memory access is more efficient when the loops are interchanged, as in Example 7-7.

Example 7-7 Simple Loop Nest Interchanged for Stride-1 Access

```
for (j=0; j<m; j++) {
   for (i=0; i<n; i++) {
      a[j][i] = 0.0;
   }
}
```

This transformation is carried out automatically by the loop nest optimizer. However, the LNO has to consider more than just the cache behavior. What if, in Example 7-6, *n*=2 and *m*=100? (Or 1000, or 10000?) Then the array fits in cache, and the original loop order achieves full cache reuse. Worse, interchanging the loops puts the shorter *i* loop inside. Software pipeline code for any loop incurs substantial overhead; pipeline code for a loop with few iterations results, essentially, in all setup and no pipeline. It would be wrong of the compiler to transform the loops in this case.

Example 7-8 shows a different problem case.

Example 7-8 Loop Nest with Data Recursion

```
for (i=0; i<n; i++) {
   for (j=0; j<m; j++) {
      a[j][i] = a[j][i] + a[j][i-1];
   }
}
```

Reversing the order of this loop nest produces nice cache behavior, but the recursion in *i* limits the performance that software pipelining can achieve for the inner loop. The LNO needs to consider the cache effects, the instruction scheduling, and loop overhead when deciding whether it should reorder such a loop nest. (In this case, the LNO assumes that *m* and *n* are large enough that optimizing for cache behavior is the best choice.)

If you know that the LNO has made the wrong loop interchange choice, you can instruct it to make a correction. Similar to the need to tell the compiler about the aliasing model (see "Understanding Aliasing Models" on page 109), you can provide the LNO with the information it needs to make the right optimization. There are two ways to supply this extra information:

- Turn off loop interchange.
- Tell the compiler which order you want the loops in.

You can turn off loop interchange for the entire module with the *-LNO:interchange=off* flag. To turn it off for a single loop nest, precede the nest with a directive or pragma:

```
#pragma no interchange
c*$* no interchange
```

To specify a particular order for a loop nest, precede the loop nest with the following directive:

```
#pragma interchange (i, j [,k ...])
c*$* interchange (i, j [,k ...])
```

This directive instructs the compiler to try to order the loops so that the loop using index variable *i* is outermost, the one using *j* is inside it, and so on. The loops should be perfectly nested (that is, there should be no code between the `do` or `for` statements). The compiler will try to order the loops as requested, but it is not guaranteed.

Combining Loop Interchange and Loop Unrolling

Loop interchange and outer loop unrolling can be combined to solve some performance problems that neither technique can solve on its own. For example, consider the data-recursive loop nest in Example 7-8.

Interchange of this loop improves cache performance, but the recurrence on *i* limits software pipeline performance. If the *j*-loop is unrolled after interchange, the recurrence will be mitigated because there will be several independent streams of work that can be used to fill up the CPU's functional units. A simplified version of the interchanged, unrolled loop is shown in Example 7-9.

Example 7-9 Recursive Loop Nest Interchanged and Unrolled

```
for (j=0; j<m; j+=4) {
   for (i=0; i<n; i++) {
      a[j+0][i] = a[j+0][i] + a[j+0][i-1];
      a[j+1][i] = a[j+1][i] + a[j+1][i-1];
      a[j+2][i] = a[j+2][i] + a[j+2][i-1];
      a[j+3][i] = a[j+3][i] + a[j+3][i-1];
   }
}
```

The LNO considers interchange, unrolling, and blocking together to achieve the overall best performance.

Controlling Cache Blocking

The idea of cache blocking is explained under "Understanding Cache Blocking" on page 149. The LNO performs cache blocking automatically. Thus a matrix multiplication like the one in Example 7-10 is automatically transformed into a blocked loop nest.

Example 7-10 Matrix Multiplication in C

```
for (j=0; j<n; j++) {
   for (i=0; i<m; i++) {
      for (k=0; k<l; k++) {
         c[i][j] += a[i][k]*b[k][j];
      }
   }
}
```

The transformation of Example 7-10, as reported in the *w2c* file (see "Reading the Transformation File" on page 158), resembles Example 7-11.

Example 7-11 Cache-Blocked Matrix Multiplication

```
for (ii=0; ii<m; ii+=b1) {
   for (kk=0; kk<l; kk+=b2) {
      for (j=0; j<n; j++) {
         for (i=ii; i<MIN(ii+b1-1,m); i++) {
            for (k=kk; i<MIN(kk+b2-1,l); k++) {
               c[i][j] += a[i][k]*b[k][j];
            }
         }
      }
   }
}
```

(The transformed code is actually more complicated than this: register blocking is used on the *i*- and *j*-loops; separate cleanup loops are generated, and the compiler takes multiple cache levels into consideration.)

Adjusting Cache Blocking Block Sizes

For the majority of programs, these transformations are the proper optimizations. But in some cases, they can hurt performance. For example, if the data already fit in the cache, the blocking merely introduces overhead, and should be avoided. Several flags and directives are available to fine-tune the blocking that the LNO performs, as summarized in Table 7-1.

Table 7-1 LNO Options and Directives for Cache Blocking

Compiler Flag	Equivalent Directive or Pragma	Effect
-LNO:blocking=off	C*$* NO BLOCKING #pragma no blocking	Prevent cache blocking of the whole module or one loop nest.
-LNO:blocking__size =[*l1*][,*l2*]	C*$* BLOCKING SIZE [*l1*][,*l2*] #pragma blocking size ([*l1*][,*l2*])	Specify block size for L1 cache, L2 cache, or both.
	C*$* BLOCKABLE (*dovar*,*dovar*...)	Specify a loop nest as blockable.

If a loop operates on in-cache data, or if you have already done your own blocking, the LNO's blocking can be turned off completely or for a specific loop nest.

Occasionally you might find a reason to dictate a block size different from the default calculated by the LNO (examine the transformation file to find out the block size it used). You can specify a specific block size for all blocked loop nests in the module with the *-LNO:blocking_size* flag, or you can specify the blocking sizes for a specific loop nest using a directive. Example 7-12 shows a Fortran loop with a specified blocking size.

Example 7-12 Fortran Nest with Explicit Cache Block Sizes for Middle and Inner Loops

```
      subroutine amat(x,y,z,n,m,mm)
      real*8 x(1000,1000), y(1000,1000), z(1000,1000)
      do k = 1, n
C*$* BLOCKING SIZE (0,200)
         do j = 1, m
C*$* BLOCKING SIZE (0,200)
            do i = 1, mm
               z(i,k) = z(i,k) + x(i,j)*y(j,k)
```

```
         enddo
      enddo
   enddo
   return
   end
```

Compiling Example 7-12, the LNO tries to make 200 x 200 blocks for the *j*- and *i*-loops. But there is no requirement that it interchange loops so that the *k*-loop is inside the other two. You can instruct the compiler that you want this done by inserting a directive with a block size of zero above the *k*-loop, as shown in Example 7-13.

Example 7-13 Fortran Loop with Explicit Cache Block Sizes and Interchange

```
      subroutine amat(x,y,z,n,m,mm)
      real*8 x(1000,1000), y(1000,1000), z(1000,1000)
C*$* BLOCKING SIZE (0,0)
      do k = 1, n
C*$* BLOCKING SIZE (0,200)
         do j = 1, m
C*$* BLOCKING SIZE (0,200)
            do i = 1, mm
               z(i,k) = z(i,k) + x(i,j)*y(j,k)
            enddo
         enddo
      enddo
      return
      end
```

This produces a blocked loop nest with a form like that shown in Example 7-14.

Example 7-14 Transformed Fortran Loop

```
      subroutine amat(x,y,z,n,m,mm)
      real*8 x(1000,1000), y(1000,1000), z(1000,1000)
      do jj = 1, m, 200
         do ii = 1, mm, 200
            do k = 1, n
               do j = jj, MIN(m, jj+199)
                  do i = ii, MIN(mm, ii+199)
                     z(i,k) = z(i,k) + x(i,j)*y(j,k)
                  enddo
               enddo
            enddo
         enddo
      enddo
      return
      end
```

Adjusting the Optimizer's Cache Model

The second way to affect block sizes is to modify the compiler's cache model. The LNO makes blocking decisions at compile time using a fixed description of the cache system of the target processor board. Although the *-Ofast=ip27* flag tells the compiler to optimize for current SN0 systems, it can't know what size the secondary cache will be in the system on which the program runs—1 MB, 4 MB, and larger secondary caches are available. As a result, it chooses to optimize for the lowest common denominator, the 1 MB cache.

A variety of flags tell the compiler to optimize for specific sizes and structures of L1, L2, and TLB caches. See the LNO(5) reference page for a discussion.

Using Loop Fusion and Fission

The Loop Nest Optimizer can transform loops by loop fusion, loop fission, and loop peeling.

Using Loop Fusion

Loop fusion, described under "Understanding Loop Fusion" on page 148, is a standard technique that can improve cache performance. The LNO performs this optimization automatically. It will even perform *loop peeling*, if necessary, to fuse loops. Example 7-15 shows a pair of adjacent loops that could be fused if they used the same range of indexes.

Example 7-15 Adjacent Loops that Cannot be Fused

```
do i = 1, n
   a(i) = 0.0
enddo
do i = 2, n-1
   c(i) = 0.5 * (b(i+1) + b(i-1))
enddo
```

The LNO applies loop peeling and then loop fusion to produce code like Example 7-16.

Example 7-16 Adjacent Loops Fused After Peeling

```
a(1) = 0.0
do i = 2, n-1
   a(i) = 0.0
   c(i) = 0.5 * (b(i+1) + b(i-1))
enddo
a(n) = 0.0
```

The first and last iterations of the first loop are peeled off so that you end up with two loops that run over the same iterations; these two loops may then be fused.

Using Loop Fission

Although loop fusion improves cache performance, it has one potential drawback: It makes the body of the loop larger. Large loop bodies place greater demands on the compiler because more registers are required to schedule the loop efficiently. The compilation takes longer since the algorithms for register allocation and software pipelining grow faster than linearly with the size of the loop body. To balance these negative aspects, the LNO also performs loop fission, whereby large loops are broken into smaller, more manageable ones. Loop fission can require the invention of new variables. Consider Example 7-17, in which an intermediate value must be saved in a temporary location *s* over several statements.

Example 7-17 Sketch of a Loop with a Long Body

```
for (j=0; j<n; j++) {
   ...
   s = ...
   ...
   ... = s
   ...
}
```

The compiler can introduce an array of temporaries and then break the loop into two pieces, as shown in Example 7-18.

Example 7-18 Sketch of a Loop After Fission

```
for (j=0; j<n; j++) {
   ...
   se[j] = ...
   ...
}
for (j=0; j<n; j++) {
   ...
   ... = se[j]
   ...
}
```

Example 7-19 shows an outer loop that controls two inner loops, neither of which uses stride-1 access. Cache behavior would be better if the loops could be reversed, but this cannot be done as the code stands.

Example 7-19 Loop Nest that Cannot Be Interchanged

```
for (j=0; j<n; j++) {
   for (i=0; i<n; i++) {
      b[i][j] = a[i][j];
   }
   for (i=0; i<n; i++) {
      c[i][j] = b[i+m][j];
   }
}
```

The LNO can apply loop fission to separate the two inner loops. Then it can interchange both, as shown in Example 7-20, and improve the cache behavior.

Example 7-20 Loop Nest After Fission and Interchange

```
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      b[i][j] = a[i][j];
   }
}
for (i=0; i<n; i++) {
   for (j=0; j<n; j++) {
      c[i][j] = b[i+m][j];
   }
}
```

Controlling Fission and Fusion

Like most optimizations, loop fission presents some trade-offs. Some common subexpression elimination that occurs when all the code is in one loop can be lost when the loops are split. In addition, more loops mean more code, so instruction cache performance can suffer. Loop fission also needs to be balanced with loop fusion, which has its own benefits and liabilities.

The compiler attempts to mix both optimizations, as well as the others it performs, to produce the best code possible. Different mixtures of optimizations are required in different situations, so it is impossible for the compiler always to achieve the best performance. As a result, several flags and directives are provided to control loop fusion and fission, as listed in the LNO(5) reference page and summarized in Table 7-2.

Table 7-2 LNO Options and Directives for Loop Transformation

Compiler Flag	Equivalent Directive or Pragma	Effect
-LNO:fission={0|1|2}		Disable fission, use it, or apply it before fusion.
-LNO:fusion={0|1|2}		Disable fusion, use it, or apply it before fission.
-LNO:fusion__peeling__limit=n		Limit loop peeling before fusion.
	C*$* FISSION #pragma fission	Request fissioning of a loop.
	C*$* FISSIONABLE #pragma fissionable	Request fission without checks.
	C*$* NO FISSION #pragma no fission	Prevent fissioning of a loop.
	C*$* FUSE #pragma fuse	Request fusing following loops.
	C*$* FUSABLE #pragma fusable	Request fusing without checks.
	C*$* NO FUSION #pragma no fusion	Prevent fusing loops.

Using Prefetching

As described under "Understanding Prefetching" on page 137, the MIPS IV ISA contains prefetch instructions—instructions that move data from main memory into cache in advance of their use. This allows some or all of the time required to access the data to be hidden behind other work. One optimization the LNO performs is to insert prefetch instructions into your program. To see how this works, consider the simple reduction loop in Example 7-21.

Example 7-21 Simple Reduction Loop Needing Prefetch

```
for (i=0; i<n; i++) {
   a += b[i];
}
```

If the loop in Example 7-21 is executed exactly as written, every secondary cache miss of the vector *b* stalls the CPU, because there is no other work to do. Now consider the modification in Example 7-22. In this example, "prefetch" is not a C statement; it indicates only that a prefetch instruction for the data at the specified address is issued at that point in the program.

Example 7-22 Simple Reduction Loop with Prefetch

```
for (i=0; i<n; i++) {
   prefetch b[i+16];
   a += b[i];
}
```

If *b* is an array of doubles, the address `b[i+16]` is one L2 cache line (128 bytes) ahead of the value that is being read in the current iteration. With this prefetch instruction inserted in the loop, each cache line is requested 16 iterations before it needs to be used. Prefetch instructions are ignored when they address a cache line that is already in-cache or on the way to cache.

Each iteration of the loop in Example 7-22 takes two cycles: the prefetch instruction and the load of `b[i]` each take one cycle and the addition is overlapped. Therefore, cache lines are prefetched 32 cycles (16, 2-cycle iterations) in advance of their use. The latency of a cache miss to local memory is approximately 60 cycles, and roughly half of this time will be overlapped with work on the previous cache line.

Prefetch Overhead and Unrolling

Example 7-22 issues a prefetch instruction for every load. For out-of-cache data, this is not bad because the time spent executing prefetch instructions would otherwise be spent waiting for the cache miss. However, if the loop processed only in-cache data, the performance would be half what it could be—half the instruction cycles would be needless prefetch instructions.

Only one prefetch instruction is needed per cache line, so it would be better to make prefetching conditional, as suggested in Example 7-23.

Example 7-23 Reduction with Conditional Prefetch

```
for (i=0; i<n; i++) {
   if ((i % 16) == 0) prefetch b[i+16];
   a += b[i];
}
```

This implementation issues just one prefetch instruction per cache line, but the instructions to implement the if-test costs more than issuing a single redundant prefetch, so it is no improvement. A better way to reduce prefetch overhead is to unroll the loop as shown in Example 7-24.

Example 7-24 Reduction with Prefetch Unrolled Once

```
for (i=0; i<n; i+=2) {
   prefetch b[i+16];
   a += b[i+0];
   a += b[i+1];
}
```

In Example 7-24, prefetch instructions are issued only eight times per cache line, so the in-cache case pays for eight fewer redundant prefetch instructions. Further unrolling reduces them even more; in fact, unrolling by 16 eliminates all redundant prefetches (and drastically reduces the loop-management overhead).

The LNO automatically inserts prefetch instructions and uses unrolling to limit the number of redundant prefetch instructions it generates. It does not unroll the loop enough to eliminate the redundancies completely because the extra unrolling, like too much loop fusion, can have a negative effect on software pipelining. So, in prefetching, as in all its other optimizations, the LNO tries to achieve a delicate balance.

The LNO does one further thing to improve prefetch performance. Instead of prefetching just one cache line ahead, it prefetches two or more cache lines ahead so that most, if not all, of the memory latency is overlapped. Consider Example 7-25, a reduction loop unrolled with a two-ahead prefetch.

Example 7-25 Reduction Loop Unrolled with Two-Ahead Prefetch

```
for (i=0; i<n; i+=8) {
   prefetch b[i+32];
   a += b[i+0];
   a += b[i+1];
   a += b[i+2];
   a += b[i+3];
   a += b[i+4];
   a += b[i+5];
   a += b[i+6];
   a += b[i+7];
}
```

This code prefetches two cache lines ahead, and makes only one redundant prefetch per cache line.

Using Pseudo-Prefetching

Prefetch instructions can be used to move data into the primary cache from either main memory or the secondary cache. However, the latency for fetching from L2 to L1 cache is small, eight to ten cycles, and it can usually be hidden without resorting to prefetch instructions. Consider Example 7-26, a reduction loop unrolled four times.

Example 7-26 Reduction Loop Unrolled Four Times

```
for (i=0; i<n; i+=4) {
   a += b[i+0];
   a += b[i+1];
   a += b[i+2];
   a += b[i+3];
}
```

Introduce a temporary register and reorder the instructions to produce Example 7-27.

Example 7-27 Reduction Loop Reordered for Pseudo-Prefetching

```
for (i=0; i<n; i+=4) {
   t  = b[i+3];
   a += b[i+0];
   a += b[i+1];
   a += b[i+2];
   a += t;
}
```

Written this way, data in the next 32-byte L1 cache line, b[i+4], is referenced several cycles before it is needed. The fetch of that line is overlapped with operations on the current cache line.

This technique, called *pseudo-prefetch*, is the default way that the LNO hides the latency of access to the secondary cache. No prefetch instructions are used, so it introduces no overhead for in-cache data. It does require the use of additional registers, however.

Controlling Prefetching

To provide the best performance for the majority of programs, the LNO generally that any variable whose size it cannot determine is not cache-resident. It generates prefetch instructions in loops involving these data. Sometimes—for example, as a result of cache blocking—it can determine that certain cache misses will be satisfied from the secondary cache rather than main memory. For these misses the LNO uses pseudo-prefetching.

Although this default behavior works well most of the time, it may impose overhead in cases where the data fit in the cache, or where you have already put your own cache blocking into the program. Several flags and directives are provided to fine-tune such situations. They are summarized in Table 7-3.

Table 7-3 LNO Options and Directives for Prefetch

Compiler Flag	Related Directive	Effect
-LNO:pf*n*=	*C*$* PREFETCH() #pragma prefetch	Enable or disable prefetching for cache level *n*.
-LNO:prefetch=*n*	(none)	Set no, normal, or aggressive prefetch.
-LNO:prefetch__ahead=*n*	(none)	How many cache lines ahead to prefetch.
-LNO:prefetch__manual	* C*$* PREFETCH__MANUAL() #pragma prefetch_manual	Enable or disable manual prefetch.
(none)	* C*$* PREFETCH__REF #pragma prefetch_ref	Generate a prefetch instruction (manual prefetch)
(none)	* C*$* PREFETCH__REF__DISABLE #pragma prefetch_ref_disable	Disable prefetching for a specified array.

Refer to the LNO(5) reference page for the compiler flags and Fortran directives, and to the *MIPSpro C and C++ Pragmas* manual listed under "Related Manuals" on page xxix for the pragmas.

Using Manual Prefetching

For standard vector code, automatic prefetching does a good job; but it can't do a perfect job for all code. One limitation is that it applies only to sequential access over dense arrays. If the program uses sparse or nonrectangular data structures, it is not always possible for the compiler to analyze where prefetch instructions should be placed. If your code does not access data in a regular fashion using loops, prefetches are also not used. Furthermore, data accessed in outer loops is not prefetched.

You can insert an explicit prefetch using the prefetch_ref pragma or directive listed in Table 7-3. This directive generates a single prefetch instruction to the memory location specified. This memory location does not have to be a compile-time constant expression; it can involve any program variables, for example, an indexed array element or a pointer expression. Automatic prefetching, if it is enabled, ignores all references to this array in the loop nest containing the directive; this allows you to use manual prefetching on some data in conjunction with automatic prefetching on other data.

Ideally, the program should execute precisely one prefetch instruction per cache line. The stride option allows you to specify how often the prefetch instruction is to be issued. For example, Example 7-28 shows a Fortran fragment that uses two-ahead manual prefetch.

Example 7-28 Fortran Use of Manual Prefetch

```
c*$* prefetch_ref=a(1)
c*$* prefetch_ref=a(1+16)
      do i = 1, n
c*$* prefetch_ref=a(i+32), stride=16, kind=rd
         sum = sum + a(i)
      enddo
```

In Example 7-28, the stride=16 clause in the directive tells the compiler that it should insert a prefetch instruction only once every 16 iterations. As with automatic prefetching, the compiler unrolls the loop and places just one prefetch instruction in the loop body. Because there is a limit to the amount of unrolling that is beneficial, the compiler may choose to unroll less than specified in the stride value.

The MIPS IV prefetch instructions allow you to specify whether the prefetched line is intended to be read from or written into. For the R10000 CPU, this information can be used to stream some data through one associative set of the cache while leaving other data in the second set; this can result in fewer cache conflicts. Thus, it is a good idea to specify whether the prefetched data are to read or written. If it will be both read and written, specify that it is written.

You can specify the level in the memory hierarchy to prefetch with the level clause. Most often you need only prefetch from main memory into L2, so the default level of 2 is correct.

You can use the size clause to tell the compiler how much space in the cache will be used by the array being manually prefetched. This value helps the compiler decide which other arrays should be prefetched automatically. If the compiler believes there is a lot of available cache space, other arrays are less likely to be prefetched because there is room for them—or, at least, blocks of them—to reside in the cache. On the other hand, if the compiler believes the other arrays will not fit in the cache, it will generate code that streams them though the cache and uses prefetch instructions to improve the out-of-cache performance. So, when using the prefetch_ref directive, if you find that other arrays are not automatically being prefetched, use the size clause with a large value to encourage the compiler to generate prefetch instructions for them. Conversely, if other arrays are being prefetched and should not be, a small size value may encourage the compiler not to prefetch the other arrays.

Using Array Padding

As described under "Understanding Cache Thrashing" on page 140, an unlucky alignment of arrays can cause significant performance penalties from cache thrashing. This is particularly likely when array dimensions are a power of 2. Fortunately, the LNO automatically pads arrays to eliminate or reduce such cache conflicts. Local variables, such as those in the typical code in Example 7-29, are spaced out so that cache thrashing does not occur.

Example 7-29 Typical Fortran Declaration of Local Arrays

```
dimension a(max), b(max), c(max), d(max)
do i = 1, max
   a(i) = b(i) + c(i)*d(i)
enddo
```

Note that if you compile Example 7-29 with *-flist*, you won't see padding explicitly in the listing file.

The MIPSpro compiler also pads C global variables, and common blocks such as the following when you use the highest optimization level, *-O3* or *-Ofast*:

```
subroutine sub
common a(512,512), b(512*512)
```

The compiler does this by putting each variable in its own common block. This optimization only occurs when both of the following are true:

- All routines that use the common block are compiled at *-O3*.
- The compiler detects no inconsistent uses of the common block across routines.

If any routine that uses the common block is compiled at an optimization level lower than *-O3*, or if there are inconsistent uses, the splitting, and hence the padding, will not be performed. Instead, the linker pastes the common blocks back together, and the original unsplit version is used. Thus, this implementation of common block padding is perfectly safe, even if you make a mistake in usage.

One programming practice causes problems. If out-of-bounds array accesses are made, wrong answers can result. For example, one improper but not uncommon usage is to operate on all data in a common block as if they were contiguous, as in Example 7-30.

Example 7-30 Common, Improper Fortran Practice

```
subroutine sub
common a(512,512), b(512*512)
do i = 1, 2*512*512
   a(i) = 0.0
enddo
end
```

In this example, both *a* and *b* are zeroed out by indexing only through *a*, under the assumption that array *b* immediately follows *a* in memory. (Note that this is not legal Fortran.) This code will not work as expected if common block padding is used. There are three ways to deal with problems of this type:

- Compile routines using this common block at *-O2* instead of *-O3*. This is not recommended because it is likely to hurt performance.
- Compile using the flag *-OPT:reorg_common=off*. This disables the common block padding optimization. Although it compensates for the improper coding style, it means other common blocks are also not padded and may result in less than optimal performance.
- Fix the code so that it is legal Fortran.

Naturally, the last alternative is recommended. To detect such programming mistakes, use the *-check_bounds* flag. Be sure to test your Fortran programs with this flag if you encounter incorrect answers only after compiling with *-O3*.

If you use interprocedural analysis, the compiler can also perform padding inside common block arrays. For example, in a common block declared as follows, the compiler pads the first dimension as part of IPA optimization:

```
common a(1024,1024)
```

Using Gather-Scatter and Vector Intrinsics

The LNO performs two other optimizations on loops: *gather-scatter* and converting scalar math intrinsics to vector calls.

Understanding Gather-Scatter

The gather-scatter optimization is best explained via an example. In the code fragment in Example 7-31, the loop can be software pipelined even though there is a branch inside it; the resulting code, however, is not fast.

Example 7-31 Fortran Loop to which Gather-Scatter Is Applicable

```
subroutine fred(a,b,c,n)
real*8 a(n), b(n), c(n)
do i = 1, n
   if (c(i) .gt. 0.0) then
      a(i) = c(i)/b(i)
      c(i) = c(i)*b(i)
      b(i) = 2*b(i)
   endif
enddo
return
end
```

Code of this type has two sources of inefficiency. First, the fact that some array elements are skipped makes it impossible for the software pipeline to work efficiently. Second, each time the `if` condition fails, the CPU performs a few instructions speculatively before the failure is evident. CPU cycles are wasted on these instructions and on internal resynchronization.

A faster implementation separate s the scan of the array and the processing of selected elements into two loops, as shown in Example 7-32.

Example 7-32 Fortran Loop with Gather-Scatter Applied

```
do i = 1, n
   deref_gs(inc_0 + 1) = i
   if (c(i) .GT. 0.0) then
      inc_0 = (inc_0 + 1)
   endif
enddo
```

```
do ind_0 = 0, inc_0 + -1
   i_gs = deref_gs(ind_0 + 1)
   a(i_gs) = (c(i_gs)/b(i_gs))
   c(i_gs) = (b(i_gs)*c(i_gs))
   b(i_gs) = (b(i_gs)*2.0)
   endif
enddo
```

Here, *deref_gs* is a scratch array allocated on the stack by the compiler.

This transformed code first scans through the data, gathering the indices for which the conditional test is true. Then the second loop does the work on the gathered data. The condition is true for all these iterations, so the `if` statement can be removed from the second loop. The loop can now be efficiently pipelined and unused speculative instructions are avoided.

By default, the compiler performs this optimization for loops that have non-nested `if` statements. For loops that have nested `if` statements, the optimization can be tried by using the flag *-LNO:gather_scatter=2*. For these loops, the optimization is less likely to be advantageous, so this is not the default. In some instances, this optimization could slow down the code (for example, if the loop iterates over a small range). In such cases, disable this optimization completely with the flag *-LNO:gather_scatter=0*.

Vector Intrinsics

In addition to the gather-scatter optimization, the LNO also converts scalar math intrinsic calls into vector calls so they can take advantage of the vector math routines in the math library (see “Standard Math Library” on page 49).

Example 7-33 Fortran Loop That Processes a Vector

```
subroutine vfred(a)
real*8 a(200)
do i = 1, 200
   a(i) = a(i) + cos(a(i))
enddo
return
end
```

The loop in Example 7-33 applies a scalar math function, **cos()**, to a stride-1 vector. The LNO can convert it into a call on the vector routine **vcos()** similar to the code in Example 7-34.

Example 7-34 Fortran Loop Transformed to Vector Intrinsic Call

```
CALL vcos$(a(1),deref_se1_F8(1),%val((201-1)),%val(1),%val(1))
DO i = 1, 200, 1
  a(i) = (a(i) + deref_se1_F8(i))
END DO
```

As documented in the math(3) reference page, the input and output arrays of the vector routines must not overlap. The code in Example 7-33 stores its result back into array *a*. In order to call the vector routine, the compiler creates *deref_se1_F8*, a scratch array allocated on the stack, to hold the vector output. The vector routine is enough faster than a scalar loop to make up for the overhead of copying.

The results of using a vector routine may not agree, bit for bit, with the result of the scalar routines. If for some reason it is critical to have numeric agreement precise to the last bit, disable this optimization with the flag *-LNO:vintr=off*.

The actual performance of the vector instrinsic functions is graphed in Figure 7-2. As it shows, the vector functions are always faster than the scalar functions when the vector has at least ten elements.

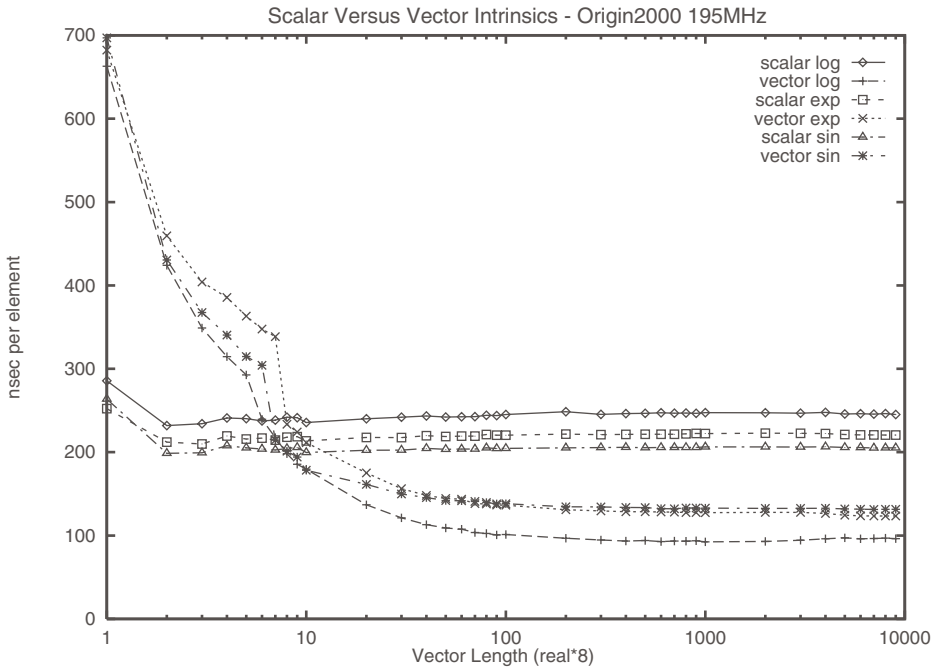


Figure 7-2 Performance of Vector Intrinsic Functions on an Origin2000

Summary

The Loop Nest Optimizer (LNO) transforms loops and nests of loops for better cache access and for more effective software pipelining. It performs many cache access modifications automatically. You can help the LNO generate the fastest code in the following ways:

- In C code, use the most aggressive aliasing model you can (see “Understanding Aliasing Models” on page 109).
- Use the ivdep directive with those loops for which it is valid (see “Breaking Other Dependencies” on page 115).
- Avoid equivalence statements in Fortran code; they introduce aliasing and prevent padding and other optimizations.

- Avoid `goto` statements; they obscure control flow and prevent the compiler from analyzing your code effectively.
- Declare and use common blocks consistently in every module so that the LNO can properly pad them.
- Don't unroll loops by hand (unless you don't want the compiler to unroll them).
- Don't violate the Fortran standard. Out-of-bounds array references can cause the common block padding optimizations to generate wrong results.

Chapter 8

Tuning for Parallel Processing

The preceding chapters have covered the process of making a program run faster in a single CPU. This chapter focusses on running the optimized program concurrently on multiple CPUs, expecting the program to complete in less time when more CPUs are applied to it. Parallel processing introduces new bottlenecks, but there are additional tools to deal with them. This chapter covers the following major topics:

- "Understanding Parallel Speedup and Amdahl's Law" on page 188 introduces the mathematical rule that limits the ultimate gain from parallel processing.
- "Compiling Serial Code for Parallel Execution" on page 193 discusses taking a one-CPU program and running it on multiple CPUs, without changing the source code.
- "Explicit Models of Parallel Computation" on page 194 contains a survey of the different types of parallel programming you can use to gain direct control of parallel execution.
- "Tuning Parallel Code for SN0" on page 196 discusses general bottlenecks that can affect any parallel program.
- "Scalability and Data Placement" on page 205 discusses the effect of data placement on performance in the SN0 distributed shared memory system, and how to tune data placement without changing program source.
- "Using Data Distribution Directives" on page 222 reviews the compiler directives you can use to specify data placement in the source of programs using the MP library.
- "Non-MP Library Programs and Dplace" on page 243 discusses data placement for programs that do not use the MP library for parallel execution.

Understanding Parallel Speedup and Amdahl's Law

There are two ways to obtain the use of multiple CPUs. You can take a conventional program in C, C++, or Fortran, and have the compiler find the parallelism that is implicit in the code. This method is surveyed under "Compiling Serial Code for Parallel Execution" on page 193.

You can write your source code to use explicit parallelism, stating in the source code which parts of the program are to execute asynchronously, and how the parts are to coordinate with each other. "Explicit Models of Parallel Computation" on page 194 is a survey of the programming models you can use for this, with pointers to the online manuals.

When your program runs on more than one CPU, its total run time should be less. But how much less? What are the limits on the speedup? That is, if you apply 16 CPUs to the program, should it finish in 1/16th the elapsed time?

Adding CPUs to Shorten Execution Time

You can distribute the work your program does over multiple CPUs. However, there is always some part of the program's logic that has to be executed serially, by a single CPU. This sets the lower limit on program run time.

Suppose there is one loop in which the program spends 50% of the execution time. If you can divide the iterations of this loop so that half of them are done in one CPU while the other half are done at the same time in a different CPU, the whole loop can be finished in half the time. The result: a 25% reduction in program execution time.

The mathematical treatment of these ideas is called Amdahl's law, for computer pioneer Gene Amdahl, who formalized it. There are two basic limits to the speedup you can achieve by parallel execution:

- The fraction of the program that can be run in parallel, p, is never 100%.
- Because of hardware constraints, after a certain point, there is less and less benefit from each added CPU.

Tuning for parallel execution comes down to battling these two limits. You strive to increase the parallel fraction, p, because in some cases even a small change in p (from 0.8 to 0.85, for example) makes a dramatic change in the effectiveness of added CPUs.

Then you work to ensure that each added CPU does a full CPU's work, and does not interfere with the work of other CPUs. In the SN0 architectures this means:

- Spreading the workload equally among the CPUs.
- Eliminating false sharing and other types of memory contention between CPUs.
- Making sure that the data used by each CPU are located in a memory near that CPU's node.

Understanding Parallel Speedup

If half the iterations of a DO-loop are performed on one CPU, and the other half run at the same time on a second CPU, the whole DO-loop should complete in half the time. For example, consider the typical C loop in Example 8-1.

Example 8-1 Typical C Loop

```
for (j=0; j<MAX; ++j) {
   z[j] = a[j]*b[j];
}
```

The MIPSpro C compiler can automatically distribute such a loop over *n* CPUs (with *n* decided at run time based on the available hardware), so that each CPU performs MAX/*n* iterations.

The speedup gained from applying *n* CPUs, Speedup(*n*), is the ratio of the one-CPU execution time to the *n*-CPU execution time: Speedup(*n*) = T(1) ÷ T(*n*). If you measure the one-CPU execution time of a program at 100 seconds, and the program runs in 60 seconds with 2 CPUs, Speedup(2) = 100 ÷ 60 = 1.67.

This number captures the improvement from adding hardware. T(*n*) ought to be less than T(1); if it is not, adding CPUs has made the program slower, and something is wrong! So Speedup(*n*) should be a number greater than 1.0, and the greater it is, the more pleased you are. Intuitively you might hope that the speedup would be equal to the number of CPUs—twice as many CPUs, half the time—but this ideal can never be achieved (well, almost never).

Understanding Superlinear Speedup

You expect Speedup(*n*) to be less than *n*, reflecting the fact that not all parts of a program benefit from parallel execution. However, it is possible, in rare situations, for Speedup(*n*) to be larger than *n*. This is called a *superlinear speedup*—the program has been sped up by more than the increase of CPUs.

A superlinear speedup does not really result from parallel execution. It comes about because each CPU is now working on a smaller set of memory. The problem data handled by any one CPU fits better in cache, so each CPU executes faster than the single CPU could do. A superlinear speedup is welcome, but it indicates that the sequential program was being held back by cache effects.

Understanding Amdahl's Law

There are always parts of a program that you cannot make parallel—code that must run serially. For example, consider the DO-loop. Some amount of code is devoted to setting up the loop, allocating the work between CPUs. This housekeeping must be done serially. Then comes parallel execution of the loop body, with all CPUs running concurrently. At the end of the loop comes more housekeeping that must be done serially; for example, if *n* does not divide MAX evenly, one CPU must execute the few iterations that are left over.

The serial parts of the program cannot be speeded up by concurrency. Let *p* be the fraction of the program's code that can be made parallel (*p* is always a fraction less than 1.0.) The remaining fraction (1-*p*) of the code must run serially. In practical cases, *p* ranges from 0.2 to 0.99.

The potential speedup for a program is proportional to *p* divided by the CPUs you can apply, plus the remaining serial part, 1-*p*. As an equation, this appears as Example 8-2.

Example 8-2 Amdahl's law: Speedup(n) Given p

$$\text{Speedup}(n) = \frac{1}{(p/n) + (1-p)}$$

Suppose $p = 0.8$; then Speedup(2) = 1 / (0.4 + 0.2) = 1.67, and
Speedup(4) = 1 / (0.2 + 0.2) = 2.5. The maximum possible speedup—if you could apply an infinite number of CPUs—would be 1 / (1-*p*). The fraction *p* has a strong effect on the possible speedup, as shown in the graph in Figure 8-1.

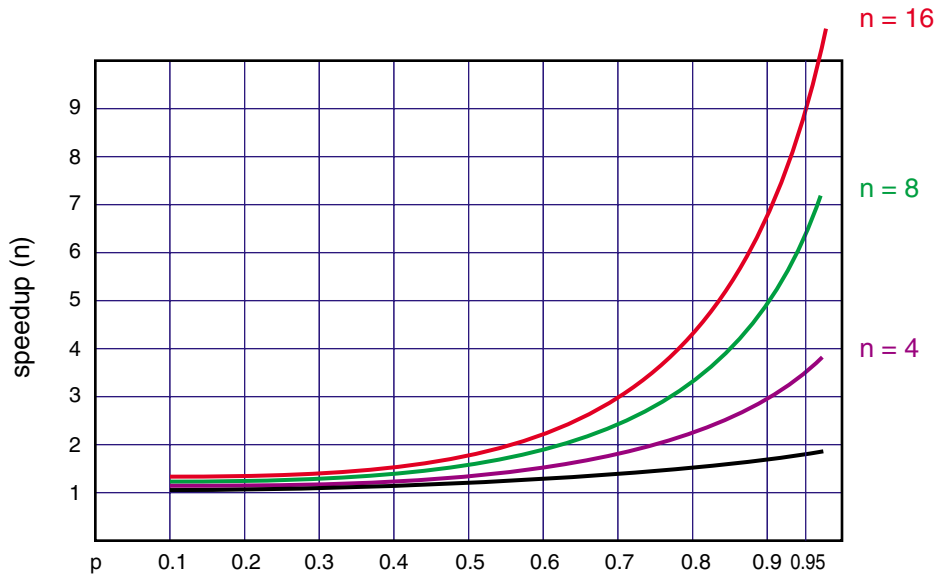


Figure 8-1 Possible Speedup for Different Values of p

Two points are clear from Figure 8-1: First, the reward for parallelization is small unless *p* is substantial (at least 0.8); or to put the point another way, the reward for increasing *p* is great no matter how many CPUs you have. Second, the more CPUs you have, the more benefit you get from increasing *p*. Using only four CPUs, you need only *p*= 0.7 to get half the ideal speedup. With eight CPUs, you need *p*= 0.85 to get half the ideal speedup.

Calculating the Parallel Fraction of a Program

You do not have to guess at the value of *p* for a given program. Measure the execution times T(1) and T(2) to calculate a measured Speedup(2) = T(1) / T(2). The Amdahl's law equation can be rearranged to yield *p* when Speedup(2) is known, as in Example 8-3.

Example 8-3 Amdahl's law: p Given Speedup(2)

$$p = \frac{2}{1} * \frac{SpeedUp(2) - 1}{SpeedUp(2)}$$

Suppose you measure T(1) = 188 seconds and T(2) = 104 seconds.

```
SpeedUp(2) = 188/104 = 1.81
p = 2 * ((1.81-1)/1.81) = 2*(0.81/1.81) = 0.895
```

In some cases, the Speedup(2) = T(1)/T(2) is a value greater than 2; in other words, a superlinear speedup ("Understanding Superlinear Speedup" on page 190). When this occurs, the formula in Example 8-3 returns a value of *p* greater than 1.0, clearly not useful. In this case you need to calculate *p* from two other more realistic timings, for example T(2) and T(3). The general formula for *p* is shown in Example 8-4, where *n* and *m* are the two CPU counts whose speedups are known, *n*>*m*.

Example 8-4 Amdahl's Law: p Given Speedup(n) and Speedup(m)

$$p = \frac{\text{Speedup}(n) - \text{Speedup}(m)}{(1 - 1/n)*\text{Speedup}(n) - (1 - 1/m)*\text{Speedup}(m)}$$

Predicting Execution Time with n CPUs

You can use the calculated value of *p* to extrapolate the potential speedup with higher numbers of CPUs. For example, if *p*=0.895 and T(1)=188 seconds, what is the expected time with four CPUs?

```
Speedup(4)= 1/((0.895/4)+(1-0.895)) = 3.04
T(4)= T(1)/Speedup(4) = 188/3.04 = 61.8
```

The calculation can be made routine using the computer. Example C-8 on page 300 shows an *awk* script that automates the calculation of *p* and extrapolation of run times.

These calculations are independent of most programming issues such as language, library, or programming model. They are not independent of hardware issues, because Amdahl's law assumes that all CPUs are equal. At some level of parallelism, adding a CPU no longer affects run time in a linear way. For example, in the Silicon Graphics POWER CHALLENGE architecture, cache friendly codes scale closely with Amdahl's law up to the maximum number of CPUs, but scaling of memory intensive applications slows as the system bus approaches saturation. When the bus bandwidth limit is reached, the actual speedup is less than predicted.

In the SN0 architecture, the situation is different and better. Some benchmarks on SN0 scale very closely to Amdahl's law up to the maximum number of CPUs, *n* = 128. However, remember that there are two CPUs per node, so some applications (in particular, applications with high requirements for local memory bandwidth) follow Amdahl's law on a per-node basis rather than a per-CPU basis. Furthermore, not all added CPUs are equal because some are farther removed from shared data and thus may have a greater latency to access that data. In general, when you can place the data used by a CPU in the same node or a neighboring node, the difference in latencies is slight and the program speeds up in line with the prediction of Amdahl's law.

Compiling Serial Code for Parallel Execution

Fortran and C programs that are written in a straightforward, portable way, without explicit source directives for parallel execution, can be parallelized automatically by the compiler. Automatic parallelization is a separately priced feature of the MIPSpro compilers beginning with version 7.2. Its use is covered in detail in the *MIPSpro Auto-Parallelizing Option Programmer's Guide* listed under "Related Manuals" on page xxix.

Compiling a Parallel Version of a Program

When the compiler option is installed, you produce a parallelized program by simply including the *-apo* compiler flag on a compile. The compiler analyzes the code for loops that can be executed in parallel, and inserts code to run those loops on multiple CPUs.

You can insert high-level compiler directives that assist the analyzer in modifying the source code. These directives are covered in the *MIPSpro Auto-Parallelizing Option Programmer's Guide.*

The compiler can produce a report showing which loops it could parallelize and which it could not parallelize, and why. The compiler can also produce a listing of the modified source code, after parallelizing and before loop-nest optimization.

Controlling a Parallelized Program at Run Time

The parallel version of the program can be run using from one CPU to as many CPUs as are available. The number of CPUs, and some other choices, are controlled externally to the program by setting environment variables. The number of CPUs the program will use is established by the value of the variable MP_SET_NUMTHREADS.

Run-time control of a parallelized program is the responsibility of *libmp*, the multiprocessing library that is automatically linked with a parallelized program. The features of libmp are covered in the mp(3) reference page. There are two versions of mp(3), one for Fortran and one for C; they are very similar except for function call syntax. This is because there is only a single library that serves programs in both languages.

Explicit Models of Parallel Computation

You can use a variety of programming models to express parallel execution in your source program. This topic summarizes the models in order to provide background for tuning, For details, see the *Topics in IRIX Programming* manual, listed in "Related Manuals" on page xxix, which contains an entire section on the topic of "models of parallel computation."

Fortran Source with Directives

Your Fortran program can contain directives that request parallel execution. When these directives are present in your program, the compiler works in conjunction with the MP runtime library, *libmp*, to run the program in parallel. There are three families of directives:

- The OpenMP (OMP) directives permit you to specify general parallel execution.

  Using OMP directives, you can write any block of code as a parallel region to be executed by multiple CPUs concurrently. You can specify parallel execution of the body of a DO-loop. Other directives coordinate parallel threads, for example, to define critical sections.

  The OMP directives are documented in the manuals listed under "Compiler Manuals" on page xxix.

- The C$DOACROSS directive and related directives, still supported for compatibility, permit you to specify parallel execution of the bodies of specified DO-loops.

  Using C$DOACROSS you can distribute the iterations of a single DO-loop across multiple CPUs. You can control how the work is divided. For example, the CPUs can do alternate iterations, or each CPU can do a block of iterations.

- The data distribution directives such as C$DISTRIBUTE and the affinity clauses added to C$DOACROSS permit you to explicitly control data placement and affinity. These have an effect only when executing on SN0 systems.

  These directives complement C$DOACROSS, making it easy for you to distribute the contents of an array in different nodes so that each CPU is close to the data it uses in its iterations of a loop body. The data distribution directives and the affinity clauses are discussed under "Using Data Distribution Directives" on page 222.

C and C++ Source with Pragmas

Your C or C++ program can contain pragma statements that specify parallel execution. These pragmas are documented in detail in the *MIPSpro C and C++ Pragmas* manual listed in "Compiler Manuals" on page xxix.

The C pragmas are similar in concept to the OMP directives for Fortran. (OpenMP directives for C and C++ are under development and will be supported in a later version of the compiler.) You use the pragmas to mark off a block of code as a parallel region. You can specify parallel execution of the body of a for-loop. Within a parallel region, you can mark off statements that must be executed by only one CPU at a time; this provides the equivalent of a critical section.

The data distribution directives and affinity clauses, which are available for Fortran, are also implemented for C in version 7.2 of the MIPSpro compilers. They are discussed under "Using Data Distribution Directives" on page 222.

Message-Passing Models MPI and PVM

There are two standard libraries, each designed to solve the problem of distributing a computation across not simply many CPUs but across many systems, possibly of different kinds. Both are supported on SN0 servers.

The MPI (Message-Passing Interface) library is designed and standardized at Argonne National Laboratory, and is documented on the MPI home page at http://www.mcs.anl.gov/mpi/index.html. The PVM (Portable Virtual Machine) library is designed and standardized at Oak Ridge National Laboratory, and is documented on the PVM home page at http://www.epm.ornl.gov/pvm/.

The Silicon Graphics implementation of the MPI library generally offers better performance than the Silicon Graphics implementation of PVM, and MPI is the recommended library. The use of these libraries is documented in the Message Passing Toolkit manuals listed in the section "Software Tool Manuals" on page xxx.

C Source Using POSIX Threads

You can write a multithreaded program using the POSIX threads model and POSIX synchronization primitives (POSIX standards 1003.1b, threads, and 1003.1c, realtime facilities). The use of these libraries is documented in *Topics in IRIX Programming*, listed in the section "Software Tool Manuals" on page xxx.

Through IRIX 6.4, the implementation of POSIX threads creates a certain number of IRIX processes and uses them to execute the pthreads. Typically the library creates fewer processes than the program creates pthreads (called an "m-on-n" implementation). You cannot control or predict which process will execute the code of any pthread at any time. When a pthread blocks, the process running it looks for another pthread to run.

Starting with IRIX 6.5, the pthreads library allocates a varying number of execution resources (basically, CPUs) and dispatches them to the runnable threads. These execution resources are allocated and dispatched entirely in the user process space, and do not require the creation of UNIX processes. As a result, pthread dispatching is more efficient.

C and C++ Source Using UNIX Processes

You can write a multiprocess program using the IRIX **sproc()** system function to create a share group of processes that execute in a single address space. Alternatively, you can create a program that uses the UNIX model of independent processes that share portions of address space using the **mmap()** system function. In either case, IRIX offers a wide variety of mechanisms for interprocess communication and synchronization.

The use of the process model and shared memory arenas is covered in *Topics in IRIX Programming* (see "Software Tool Manuals" on page xxx) and in the and sproc(2), mmap(2) and usinit(3) reference pages.

Tuning Parallel Code for SN0

Parallelizing a program is a big topic, worthy of a book all on its own. In fact, there are several good books and online courses on the subject (see "Third-Party Resources" on page xxx for a list of some). Parallel programming is a very large topic, and this guide does not attempt to teach it. It assumes that you are already familiar with the basics and concentrates on what is different about the SN0 architecture.

Prescription for Performance

Of course, what's new about SN0 is its novel shared memory architecture (see "Understanding Scalable Shared Memory" on page 6). The change in architecture has the following implications:

1. You don't have to program differently for SN0 than for any other shared memory computer. In particular, binaries for earlier Silicon Graphics systems run on SN0, in most cases, with very good performance.
2. When programs do not scale as well as they should, simply taking account of the physically distributed memory usually restores optimum performance. This can often be done outside the program, or with only simple source changes.

The basic prescription for tuning a program for parallel execution is as follows:

1. Tune for single-CPU performance, as covered at length in Chapters 4 through 7.
2. Make sure the program is fully and properly parallelized; the parallel fraction *p* controls the effectiveness of added hardware (see "Calculating the Parallel Fraction of a Program" on page 191).
3. Examine cache use, and eliminate cache contention and false sharing.
4. Examine memory access, and apply the right page-placement method.

These steps are covered in the rest of this chapter.

Ensuring That the Program Is Properly Parallelized

The first step in tuning a parallel program is making sure that it has been properly parallelized. This means, first, that enough of the program can be parallelized to allow the program to attain the desired speedup. Use the Amdahl's law extrapolation to determine the parallel fraction of the code ("Calculating the Parallel Fraction of a Program" on page 191). If the fraction is not high enough for effective scaling, there is no point in further tuning until it has been increased. For a program that is automatically parallelized, work with the compiler to remove dependencies and increase the parallelized portions (see "Compiling a Parallel Version of a Program" on page 193). When the program uses explicit parallelization, you must work on the program's design, a subject beyond the scope of this book (see "Third-Party Resources" on page xxx).

Proper parallelization means, second, that the workload is distributed evenly among the CPUs. You can use SpeedShop to profile a parallel program; it provides information on each of the parallel threads. You can use this information to verify that each thread takes about the same amount of time to carry out its pieces of the work. If this is not so, some CPUs are idling with no work at some times. In a program that is parallelized using Fortran or C directives, it may be possible to get better balance by changing the loop scheduling (such as cyclic, dynamic, or gss). In other programs it may again require algorithmic redesign.

If the program has run successfully on another parallel system, both these issues have been addressed. But if you are now running the program on more CPUs previously available, it is still possible to encounter problems in the parallelization that simply never showed up with lesser parallelism. Be sure to revalidate that any limits in scalability are not due to Amdahl's law, and watch for bugs in code that has not previously been stressed.

Finding and Removing Memory Access Problems

The location of data in the SN0 distributed memory is not important when the parallel threads of a program access memory primarily through the L1 and L2 caches. When there is a high ratio of cache (and TLB) hits, the relatively infrequent references to main memory are simply not a significant factor in performance. As a result, you should remove any cache-contention problems before you think about data placement in the SN0 distributed memory. (For an overview of cache issues, see "Understanding the Levels of the Memory Hierarchy" on page 135.)

You can determine how cache-friendly a program is using *perfex*; it tells you just how many primary, secondary, and TLB cache misses the program generates and what the cache hit ratios are, and it will estimate how much the cache misses cost (as discussed under "Getting Analytic Output with the -y Option" on page 56). There are several possible sources of poor scaling and they can generally be distinguished by examining the event counts returned by *perfex*:

- Load imbalance: Is each thread doing the same amount of work? One way to check this is to see if if all threads issue a similar number of floating point operations (event 21).
- Excessive synchronization cost: Are counts of store conditionals high (event 4)?
- Cache contention: Are counts of store exclusive to shared block high (event 31)?

When cache misses account for only a small percentage of the run time, the program makes good use of the memory and, more important, its performance will not be affected by data placement. On the other hand, if the time spent in cache misses at some level is high, the program is not cache-friendly. Possibly, data placement in the SN0 distributed memory could affect performance, but there are other issues, common to all cache-based shared memory systems, that are likely to be the source of performance problems, and these should be addressed first.

Diagnosing Cache Problems

If a properly parallelized program does not scale as well as expected, there are several potential causes. The first thing to check is whether some form of cache contention is slowing the program. You have presumably done this as part of single-CPU tuning ("Identifying Cache Problems with Perfex and SpeedShop" on page 142), but new cache contention problems can appear when a program starts executing in parallel.

New problems can be an issue, however, only for data that are frequently updated or written. Data that are mostly read and rarely written do not cause cache coherency contention for parallel programs.

The mechanism used in SN0 to maintain cache coherence is described under "Understanding Directory-Based Coherency" on page 13. When one CPU modifies a cache line, any other CPU that has a copy of the same cache line is notified, and discards its copy. If that CPU needs that cache line again, it fetches a new copy. This arrangement can cause performance problems in two cases:

- When one CPU repeatedly updates a cache line that other CPUs use for input, all the reading CPUs are forced to frequently retrieve new copies of the cache line from memory. This slows all the reading CPUs.
- When two or more CPUs repeatedly update the same cache line, they contend for the exclusive ownership of the cache line. Each CPU has to get ownership and fetch a new copy of the cache line before it can perform its update. This forces the updating CPUs to execute serially, as well as making all other CPUs fetch a new copy of the cache line on every use.

In some cases, these problems arise because all parallel threads in the program are genuinely contending for use of the same key global variables. In that case, the only cure is an algorithmic change.

More often, cache contention arises because the CPUs are using and updating unrelated variables that happen to fall in the same cache line. This is *false sharing*. Fortunately, IRIX tools can help you identify these problems. Cache contention is revealed when *perfex* shows a high number of cache invalidation events: counter events 31, 12, 13, 28, and 29. A CPU that repeatedly updates the same cache line shows a high count of stores to shared cache lines: counter 31. (See "Cache Coherency Events" on page 282.)

Identifying False Sharing

False sharing can be demonstrated with code like that in Example 8-5.

Example 8-5 Fortran Loop with False Sharing

```
      integer m, n, i, j
      real    a(m,n), s(m)
c$doacross local(i,j), shared(s,a)
      do i = 1, m
         s(i) = 0.0
         do j = 1, n
            s(i) = s(i) + a(i,j)
         enddo
      enddo
```

This code calculates the sums of the rows of a matrix. For simplicity, assume *m*=4 and that the code is run on up to four CPUs. What you observe is that the time for the parallel runs is longer than when just one CPU is used. To understand what causes this, look at what happens when this loop is run in parallel.

The following is a time line of the operations that are carried out (more or less) simultaneously:

	t=0	t=1	t=2	
CPU 0	`s(1) = 0.0`	`s(1) = s(1) + a(1,1)`	`s(1) = s(1) + a(1,2)`	...
CPU 1	`s(2) = 0.0`	`s(2) = s(2) + a(2,1)`	`s(2) = s(2) + a(2,2)`	...
CPU 2	`s(3) = 0.0`	`s(3) = s(3) + a(3,1)`	`s(3) = s(3) + a(3,2)`	...
CPU 3	`s(4) = 0.0`	`s(4) = s(4) + a(4,1)`	`s(4) = s(4) + a(4,2)`	...

At each stage of the calculation, all four CPUs attempt to update one element of the sum array, *s(i)*. For a CPU to update one element of *s*, it needs to gain exclusive access to the cache line holding that element, but the four words of *s* are probably contained in a single cache line, so only one CPU at a time can update an element of *s*. Instead of operating in parallel, the calculation is serialized.

Actually, it's a bit worse than merely serialized. For a CPU to gain exclusive access to a cache line, it first needs to invalidate cached copies that may reside in the other caches. Then it needs to read a fresh copy of the cache line from memory, because the invalidations will have caused data in some other CPU's cache to be written back to main memory. In a sequential version of the program, the element being updated can be kept in a register, but in the parallel version, false sharing forces the value to be continually reread from memory, in addition to serializing the updates.

This serialization is purely a result of the unfortunate accident that the different elements of *s* ended up in the same cache line. If each element were in a separate cache line, each CPU could keep a copy of the appropriate line in its cache, and the calculations could be done perfectly in parallel. This shows how to fix the problem: spread the elements of *s* out so that each one resides in its own cache line. One way to do that is shown in Example 8-6.

Example 8-6 Fortran Loop with False Sharing Removed

```
      integer m, n, i, j
      real    a(m,n), s(32,m)
c$doacross local(i,j), shared(s,a)
      do i = 1, m
         s(1,i) = 0.0
         do j = 1, n
            s(1,i) = s(1,i) + a(i,j)
         enddo
      enddo
```

The elements *s(1,1)*, *s(1,2)*, *s(1,3)* and *s(1,4)* are separated by at least $32 \times 4 = 128$ bytes, and so are guaranteed to fall in separate cache lines. Implemented this way, the code achieves perfect parallel speedup.

Note that the problem of false sharing is not specific to the SN0 architecture. It occurs in any cached-coherent shared memory system.

To see how this works for a real code, see an example from a paper presented at the Supercomputing '96 conference (read it online at http://www.supercomp.org/sc96/proceedings/SC96PROC/ZAGHA/INDEX.HTM). One example in this paper is a weather modeling program that shows poor parallel scaling (the red curve in Figure 8-2).

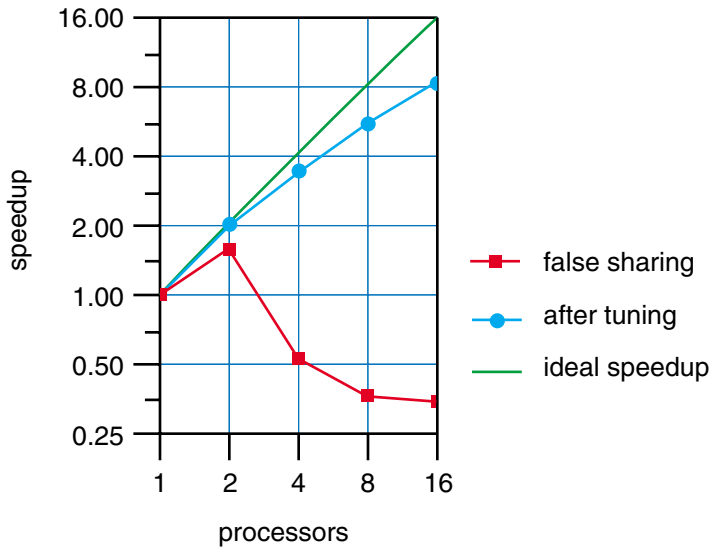


Figure 8-2 Performance of Weather Model Before and After Tuning

Running the program under *perfex* revealed that the number of secondary data cache misses (event 26) increased as the number of CPUs increased, as did the number of stores exclusive to a shared block (event 31). The large event 31 counts, increasing with the secondary cache misses, indicated a likely problem with false sharing. (The large number of secondary cache misses were a problem as well.)

The source of the problem was found using *ssrun*. There are several events that can be profiled to determine where false sharing occurs. The natural one to use is event 31, "store/prefetch exclusive to shared block in scache." There is no explicitly named experiment type for this event, so profiling it requires setting the following environment variables:

```
% setenv _SPEEDSHOP_HWC_COUNTER_NUMBER 31
% setenv _SPEEDSHOP_HWC_COUNTER_OVERFLOW 99
% ssrun -exp prof_hwc a.out
```

(See "Sampling Through Other Hardware Counters" on page 66.) You could also profile interventions (event 12) or invalidations (event 13), but using event 31 shows where the source of the problem is in the program (some thread asking for exclusive access to a cache line) rather than the place a thread happens to be when an invalidation or intervention occurs. Another event that can be profiled is secondary cache misses (event 26). For this event, use the *-dsc_hwc* experiment type and don't set any environment variables.

For this program, profiling secondary cache misses was sufficient to locate the source of the false sharing. The profile showed that the majority of secondary cache misses occurred in an accumulation step, similar to the row summation in Example 8-5. Padding was used to move each CPU's accumulation variable into its own cache line. After doing this, the performance improved dramatically (the blue curve in Figure 8-2).

Correcting Cache Contention in General

You can often deal with cache contention by changing the layout of data in the source program, but sometimes you may have to make algorithmic changes as well. If profiling indicates that there is cache contention, examine the parallel regions identified; any assignment to memory in the parallel regions is a possible source of the contention. You need to determine if the assigned variable, or the data adjacent to it, is used by other CPUs at the same time. If so, the assignment forces the other CPUs to read a fresh copy of the cache line, and this is a source of contention.

To deal with cache contention, you have following general strategies:

1. Minimize the number of variables that are accessed by more than one CPU.
2. Segregate non-volatile (rarely updated) data items into cache lines different from volatile (frequently updated) items.
3. Isolate unrelated volatile items into separate cache lines to eliminate false sharing.
4. When volatile items are updated together, group them into single cache lines.

An update of one word (that is, a 4-byte quantity) invalidates all the 31 other words in the same L2 cache line. When those other words are not related to the new data, false sharing results. Use strategy 3 to eliminate the false sharing.

Be careful when your program defines a group of global status variables that is visible to all parallel threads. In the normal course of running the program, every CPU will cache a copy of most or all of this common area. Shared, read-only access does no harm. But if items in the block are volatile (frequently updated), those cache lines are invalidated often. For example, a global status area might contain the anchor for a LIFO queue. Each time a thread puts or takes an item from the queue, it updates the queue head, invalidating that cache line.

It is inevitable that a queue anchor field will be frequently invalidated. The time cost, however, can be isolated to the code that accesses the queue by applying strategy 2. Allocate the queue anchor in separate memory from the global status area. Put only a pointer to the queue anchor (a non-volatile item) in the global status block. Now the cost of fetching the queue anchor is born only by CPUs that access the queue. If there are other items that are updated along with the queue anchor—such as a lock that controls exclusive access to the queue—place them adjacent to the queue, aligned so that all are in the same cache line (strategy 4). However, if there are two queues that are updated at unrelated times, place the anchor of each in its own cache line (strategy 3).

Synchronization objects such as locks, semaphores, and message queues are global variables that must be updated by each CPU that uses them. You may as well assume that synchronization objects are always accessed at memory speeds, not cache speeds. You can do two things to reduce contention:

- Minimize contention for locks and semaphores through algorithmic design. In particular, use more, rather than fewer, semaphores, and make each one stand for the smallest resource possible so as to minimize the contention for any one resource. (Of course, this makes it more difficult to avoid deadlocks.)
- Never place unrelated synchronization objects in the same cache line. A lock or semaphore may as well be in the same cache line as the data that it controls, because an update of one usually follows an update of the other. But unrelated locks or semaphores should be in different cache lines.

When you make a loop run in parallel, try to ensure that each CPU operates on its own distinct sections of the input and output arrays. Sometimes this falls out naturally, but there are also compiler directives for just this purpose. (These are described in “Using Data Distribution Directives” on page 222.)

Carefully review the design of any data collections that are used by parallel code. For example, the root and the first few branches of a binary tree are likely to be visited by every CPU that searches that tree, and they will be cached by every CPU. However, elements at higher levels in the tree may be visited by only a few CPUs. One option is to pre-build the top levels of the tree so that these levels never have to be updated once the program starts. Also, before you implement a balanced-tree algorithm, consider that tree-balancing can propagate modifications all the way to the root. It might be better to cut off balancing at a certain level and never disturb the lower levels of the tree. (Similar arguments apply to B-trees and other branching structures: the "thick" parts of the tree are widely cached and should be updated least often, while the twigs are less frequently used.)

Other classic data structures can cause memory contention, and algorithmic changes are needed to cure it:

- The two basic operations on a heap (also called a priority queue) are "get the top item" and "insert a new item." Each operation ripples a change from end to end of the heap-array. However, the same operations on a linked list are read-only at all nodes except for the one node that is directly affected. Therefore, a priority list used by parallel threads might be faster implemented as a linked list than as a heap—the opposite of the usual result.
- A hash table can be implemented compactly, with only a word or two in each entry. But that creates false sharing by putting several table entries (which by definition are logically unrelated) into the same cache line. Avoid false sharing: make each hash table entry a full 128 bytes, cache-aligned. (You can take advantage of the extra space to store a list of overflow hits in each entry. Such a list can be quickly scanned because the entire cache line is fetched as one operation.)

Scalability and Data Placement

Data placement issues do not arise for all parallel programs. Those that are cache friendly do not incur performance penalties even when data placement is not optimal, because such programs satisfy their memory requests primarily from cache, rather than main memory. Data placement can be an issue only for parallel programs that are memory intensive and are not cache friendly.

If a memory-intensive parallel program exhibits scaling that is less than expected, and you are sure that false sharing and other forms of cache contention are not a problem, consider data placement. Optimizing data placement is a new performance issue unique to SN0 and other distributed-memory architectures.

The IRIX operating system should automatically ensure that all programs achieve good data placement. There are two things IRIX wants to optimize:

- The program's topology; that is, the processes making up the parallel program should run on nodes that minimize access costs for data they share.
- The page placement; that is, the memory a process accesses the most often should be allocated from its own node, or the minimum distance from that node.

Accomplishing these two tasks automatically for all programs is virtually impossible. The operating system simply doesn't have enough information to do a perfect job, but it does the best it can to approximate a good solution. The policies and topology choices the operating system uses to try to optimize data placement were described under "IRIX Memory Locality Management" on page 29. IRIX uses:

- *memory locality domain*s (MLDs) and *policy module*s (PMs) to achieve a good topology ("Memory Locality Management" on page 31)
- A variety of page placement policies to try to get the initial page placements correct ("Data Placement Policies" on page 40)
- Dynamic page migration to correct placement errors that arise as a program runs ("Dynamic Page Migration" on page 30)

This technology produces good results for many, but not all, programs. It's time to discuss how you can tune data placement, if the operating system's efforts are insufficient.

Data placement is tuned by specifying the appropriate policies and topologies, and if need be, programming with them in mind. They are specified using the utility *dplace*, environment variables understood by the MP library, and compiler directives.

Tuning Data Placement for MP Library Programs

Unlike false sharing, there is no profiling technique that will tell you definitively that poor data placement is hurting the performance of your program. Poor placement is a conclusion you have to reach after eliminating the other possibilities. Fortunately, once you suspect a data placement problem, many of the tuning options are very easy to perform, and you can often allay your suspicions or solve the problem with a few simple experiments.

The techniques for tuning data placement can be separated into two classes:

- Use MP library environment variables to adjust the operating system's default data placement policies. This is simple to do, involves no modifications to your program, and is all you will have to do to solve many data placement problems.
- Modify the program to ensure an optimal data placement. This approach requires more effort, so try this only if the first approach does not work, or if you are developing a new application. The amount of effort in this approach ranges from simple things such as making sure that the program's data initializations—as well as its calculations—are parallelized, to adding some data distribution compiler directives, to modifying algorithms as would be done for purely distributed memory architectures.

Trying Round-Robin Placement

Data placement policies used by the operating system are introduced under "Data Placement Policies" on page 40. The default policy is called first-touch. Under this policy, the process that first touches (that is, writes to, or reads from) a page of memory causes that page to be allocated in the node on which the process is running. This policy works well for sequential programs and for many parallel programs as well. For example, this is just what you want for message-passing programs that run on SN0. In such programs each process has its own separate data space. Except for messages sent between the processes, all processes use memory that should be local. Each process initializes its own data, so memory is allocated from the node the process is running in, thus making the accesses local.

But for some parallel programs, the first-touch policy can have unintended side effects. As an example, consider a program parallelized using the MP library. In parallelizing such a program, the programmer worries only about parallelizing those parts of the program that take the most amount of time. Often, data initialization code takes little time, so they are not parallelized, and so are executed by the main thread of the program. Under the first-touch policy, all the program's memory ends up allocated in the node running the main thread. Having all the data concentrated in one node or within a small radius of it creates a bottleneck: all data accesses are satisfied by one hub, and this limits the memory bandwidth. If the program is run on only a few CPUs, you may not notice a bottleneck. But as you add more CPUs, the one hub that serves the memory becomes saturated, and the speed does not scale as predicted by Amdahl's law.

One easy way to test for this problem is to try other memory allocation policies. The first one you should try is round-robin allocation. Under this policy, data are allocated in a round-robin fashion from all the nodes the program runs on. Thus, even if the data are initialized sequentially, the memory holding them will not be allocated from a single node; it will be evenly spread out among all the nodes running the program. This may not place the data in their optimal locations—that is, the access times are not likely to be minimized—but there will be no bottlenecks, so scalability will not be limited.

You can enable round-robin data placement for an MP library program with the following environment variable setting:

```
% setenv _DSM_ROUND_ROBIN
```

The performance improvement this can make is demonstrated with an example. The plot in Figure 8-3 shows the performance achieved for three parallelized runs of the double-precision vector operation

```
a(i) = b(i) + q*c(i)
```

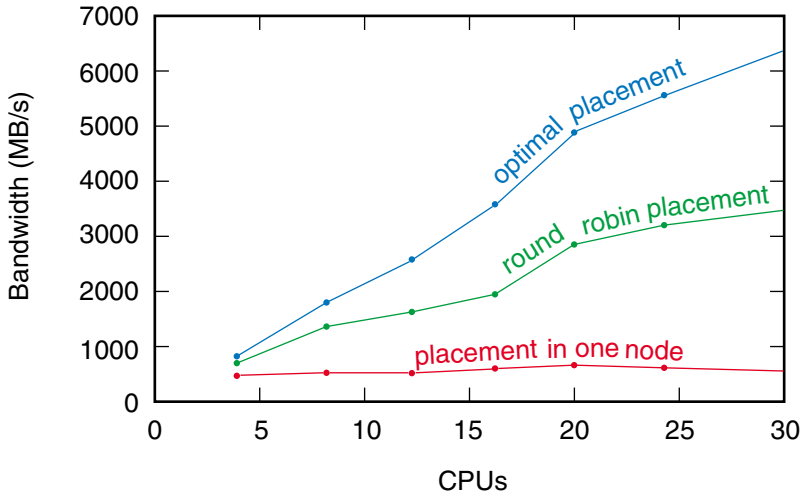


Figure 8-3 Calculated Bandwidth for Different Placement Policies

The bandwidth calculated assumes that 24 bytes are moved for each vector element. The red curve (“placement in one node”) shows the performance you attain using a first-touch placement policy and a sequential initialization so that all the data end up in the memory of one node. The memory bottleneck this creates severely limits the scaling. The green curve (“round-robin placement”) shows what happens when the data placement policy is changed to round-robin. The bottleneck disappears and the performance now scales with the number of CPUs. The performance isn’t ideal—the blue line (“optimal placement”) shows what you measure if the data are placed optimally—but by spreading the data around, the round-robin policy makes the performance respectable. (Ideal data placement cannot be accomplished simply by initializing the data in parallel.)

Trying Dynamic Page Migration

If a round-robin data placement solves the scaling problem that led you to tune the data placement, then you’re done! But if the performance is still not up to your expectations, the next experiment to perform is enabling dynamic page migration.

The IRIX automatic page migration facility is disabled by default because page migration is an expensive operation that impacts all CPUs, not just the ones used by the program whose data are being moved. You can enable dynamic page migration for a specific MP library program by setting the environment variable _DSM_MIGRATION. You can set it to either ON or ALL_ON:

```
% setenv _DSM_MIGRATION ON
% setenv _DSM_MIGRATION ALL_ON
```

When set to ALL_ON, all program data is subject to migration. When set only to ON, only those data structures that have not been explicitly placed via the compiler’s data distribution directives (“Using Data Distribution Directives” on page 222) will be migrated.

Enabling migration is beneficial when a poor initial placement of the data is responsible for limited performance. However, it cannot be effective unless the program runs for at least half a minute, first because the operating system needs some time to move the data to the best layout, and second because the program needs to execute for some time with the pages in the best locations in order to recover the time cost of migrating.

In addition to the MP library environment variables, migration can also be enabled as follows:

- For non-MP library programs, run the program using *dplace* with the *-migration* option. This is described under "Enabling Page Migration" on page 245.
- The system administrator can temporarily enable page migration for all programs using the *sn* command (see the sn(1M) reference page) or enable it permanently by using *systune* to set the numa_migr_base_enabled system parameter. (See the systune(1M) reference page, the *System Configuration* manual listed in "Software Tool Manuals" on page xxx, and the comments in the file */var/sysgen/mtune/numa*.)

Combining Migration and Round-Robin Placement

You can try to reduce the cost of migrating from a poor initial location to the optimal one by combining a round-robin initial placement with migration. You won't know whether round-robin, migration, or both together will produce the best results unless you try the different combinations, so tuning the data placement requires experimentation. Fortunately, the environment variables make these experiments easy to perform.

Figure 8-4 shows the results of several experiments on the vector operation

```
a(i) = b(i) + q*c(i)
```

In addition to combinations of the round-robin policy and migration, the effect of serial and parallel initializations are shown. For the results presented in the diagram, the vector operation was iterated 100 times and the time per iteration was plotted to see the performance improvement as the data moved toward optimal layout.

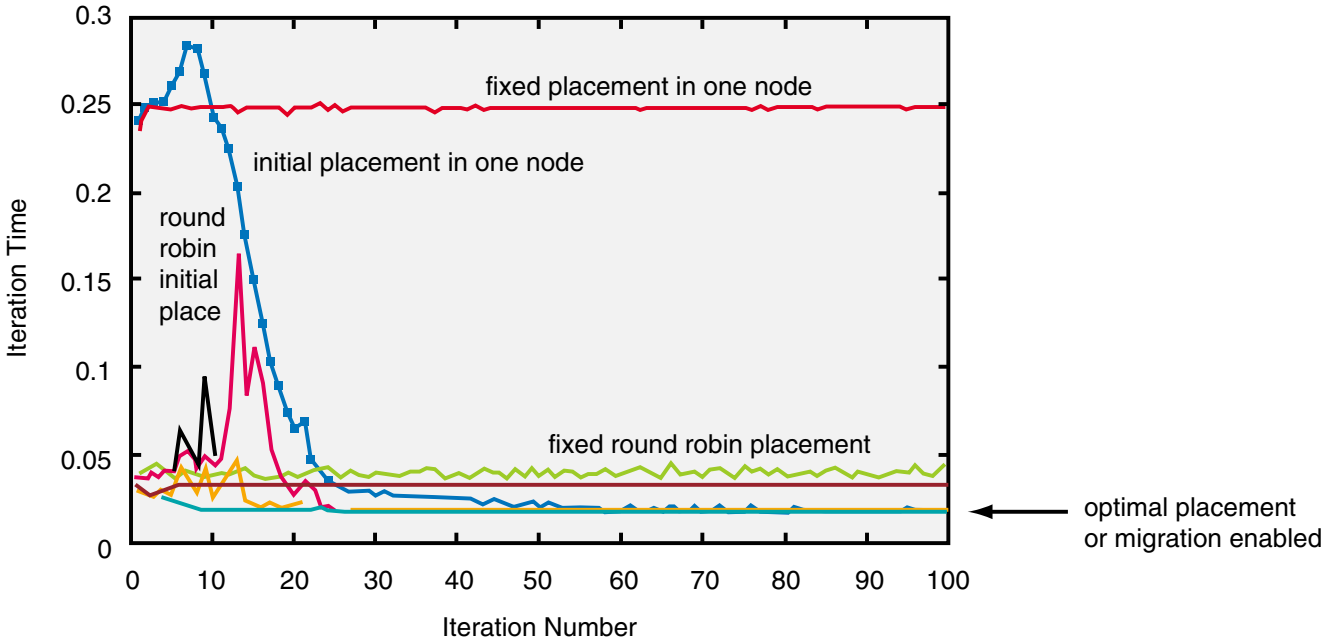


Figure 8-4 Calculated Iteration Times for Different Placement Policies

The red curve at the top (“fixed placement in one node”) shows the performance for the poor initial placement caused by a sequential data initialization with a first-touch policy. No migration is used, so this performance is constant over all iterations. The green line near the bottom (“fixed round-robin placement”) shows that a round-robin initial placement fixes most of the problems with the serial initialization: the time per iteration is five times faster than with the first-touch policy. Ultimately, though, the performance is still a factor of two slower than if an optimal data placement had been used. Once again, migration is not enabled, so the performance is constant over all iterations.

The flat curve just below the green curve shows the performance achieved when a parallel initialization is used in conjunction with a round-robin policy. Its per-iteration time is constant and nearly the same as when round-robin is used with a sequential initialization. This is to be expected: the round-robin policy spreads the data evenly among the CPUs, so it matters little how the data are initialized.

The remaining five curves all asymptote to the optimal time per iteration. Four of these eventually achieve the optimal time due to the use of migration. The turquoise curve uses a parallel initialization with a first-touch policy to achieve the optimal data placement from the outset. The orange curve just above it starts out the same, but for it, migration is enabled. The result of using migration on perfectly placed data is only to scramble the pages around before finally settling down on the ideal time. Above this curve are a magenta and another orange curve ("round-robin initial placement"). They show the effect of combining migration and a round-robin policy. The only difference is that the magenta curve used a serial initialization while the orange curve used a parallel initialization. For these runs, the serial initialization took longer to reach the steady-state performance, but you should not conclude that this is a general property; when a round-robin policy is used, it doesn't matter how the data are initialized.

Finally, the blue curve ("initial placement in one node") shows how migration improves a poor initial placement in which all the data began in one node. It takes longer to migrate to the optimal placement than the other cases, indicating that by combining a round-robin policy with migration, one can do better than by using just one of the remedies alone. Dramatic evidence of this is seen when the cumulative program run time is plotted, as shown in Figure 8-5.

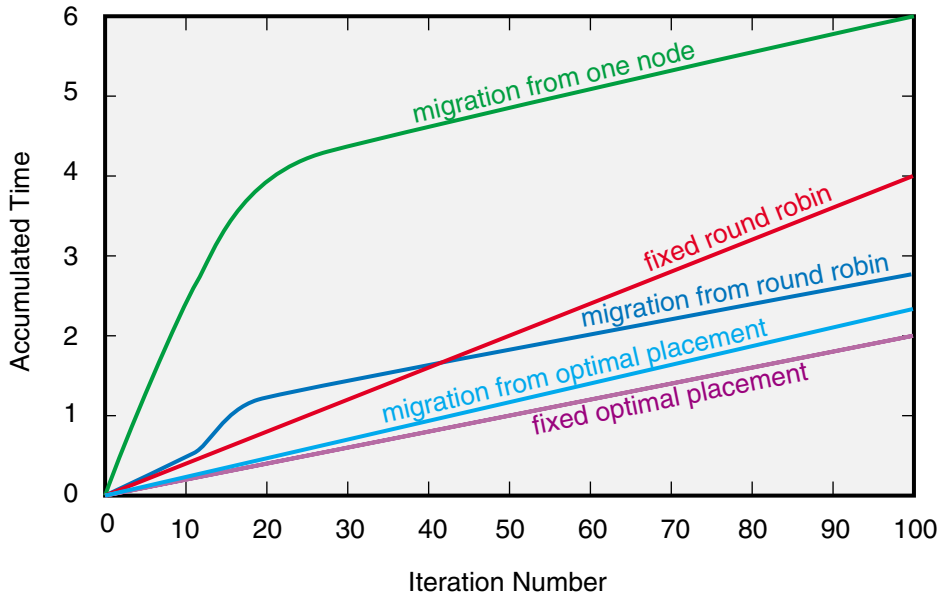


Figure 8-5 Cumulative Run Time for Different Placement Policies

These results show clearly that, for this program, migration eventually gets the data to their optimal location and that migrating from an initial round-robin placement is faster than migrating from an initial placement in which all the data are on one node. Note, though, that migration does take time: if only a small number of iterations are going to be performed, it is better to use a round-robin policy without migration.

Experimenting with Migration Levels

The results for data migration were generated using this environment variable setting:

```
% setenv _DSM_MIGRATION ON
```

(This program has no data distribution directives in it, so in this case, there is no difference between a setting of ON and one of ALL_ON.) But you can also adjust the migration level, that is, control how aggressively the operating system migrates data from one memory to another. The migration level is controlled with the following environment variable setting:

```
% setenv _DSM_MIGRATION_LEVEL level
```

A high level means aggressive migration, and a low level means nonaggressive migration. A level of 0 means no migration. The migration level can be used to experiment with how aggressively migration is performed. The plot in Figure 8-6 shows the improvement in time per iteration for various migration levels as memory is migrated from an initial placement in one node. As you would expect, with migration enabled, the data will eventually end up in an optimal location, while the more aggressive migration is, the faster the pages get to where they should be.

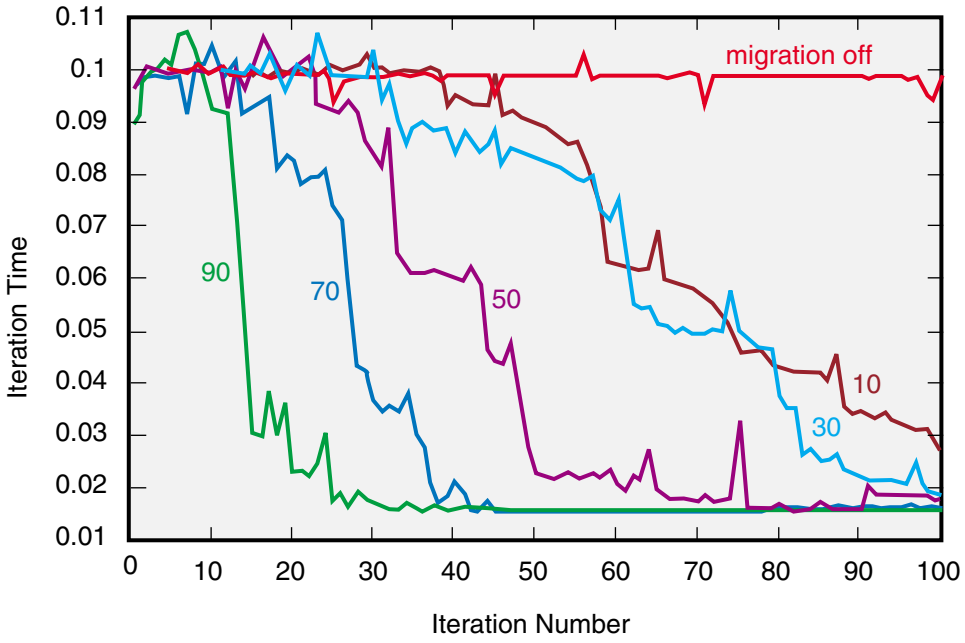


Figure 8-6 Effect of Migration Level on Iteration Time

The best level for a particular program will depend on how much time is available to move the data to their optimal location. A low migration level is fine for a long-running program with a poor initial data placement. But if a program can only afford a small amount of time for the data to redistribute, a more aggressive migration level is needed. Keep in mind that migration has an impact on system-wide performance, and the more aggressive the migration, the greater the impact. For most programs the default migration level should be sufficient.

Tuning Data Placement without Code Modification

In general, you should try the following experiments when you need to tune data placement:

1. See if using a round-robin policy fixes the scaling problem. If so, you need not try other experiments.
2. Try migration. If migration achieves about the same performance as round-robin, round-robin is to be preferred because it has less of an impact on other users of the system.
3. If migration alone achieves a better performance than round-robin alone, try the combination of both. This combination might not improve the steady-state performance, but it might get it to the steady-state performance faster.

Note that these data placement tuning experiments apply only to MP library programs:

- Sequential programs aren't affected by data placement.
- First-touch is the right placement policy for message-passing programs.
- These environment variables are understood only by the MP library (for other features of *libmp*, see the mp(3) reference page).

Modifying the Code to Tune Data Placement

In some cases you will need to tune the data placement via code modifications. There are three levels of code modification you can use:

- Rely on the default first-touch policy, and program with it in mind. This programming style is easy to use: often, fixing data placement problems is as simple as making sure all the data are initialized in a parallelized loop, so that the data starts off in the memory of the CPU that will use it.
- Insert regular data distribution directives (for example, C$DISTRIBUTE). These directives allow you to specify how individual data structures should be distributed among the memory of the nodes running the program, subject to the constraint that only whole pages can be distributed.
- Insert reshaping directives (for example, C$DISTRIBUTE_RESHAPE). These allow you to specify how individual data structures should be distributed among the memory of the nodes running the program, but they are not restricted to whole-page units. As a result, distributed data structures are not guaranteed to be contiguous; that is, they can have holes in them, and if you do not program properly, this can break your program.

Each of these three approaches requires the programmer to think about data placement explicitly. This is a new concept that doesn't apply to conventional shared-memory architectures. However, the concept has been used for years in programming distributed memory computers, and if you have written a message-passing program, you understand it well. These approaches are discussed in the three topics that follow.

Programming For First-Touch Placement

If there is a single placement of data that is optimal for your program, you can use the default first-touch policy to cause each CPU's share of the data to be allocated from memory local to its node. As a simple example, consider parallelizing the vector operation in Example 8-7.

Example 8-7 Easily Parallelized Fortran Vector Routine

```
      integer i, j, n, niters
      parameter (n = 8*1024*1024, niters = 100)
      real a(n), b(n), q
c initialization
      do i = 1, n
         a(i) = 1.0 - 0.5*i
         b(i) = -10.0 + 0.01*(i*i)
      enddo
c real work
      do it = 1, niters
         q = 0.01*it
         do i = 1, n
            a(i) = a(i) + q*b(i)
         enddo
      enddo
```

This vector operation is easy to parallelize; the work can be divided among the CPUs of a shared memory computer any way you would like. For example, if p is the number of CPUs, the first CPU can carry out the first n/p iterations, the second CPU the next n/p iterations, and so on. (This is called a simple schedule type.) Alternatively, each thread can perform one of the first p iterations, then one of the next p iterations and so on. (This is called an interleaved schedule type.)

In a cache-based machine, not all divisions of work produce the same performance. The reason is that if a CPU accesses the element *a(i)*, the entire cache line containing *a(i)* is moved into its cache. If the same CPU works on the following elements, they will be in cache. But if different CPUs work on the following elements, the cache line will have to be loaded into each one's cache. Even worse, false sharing is likely to occur. Thus performance is best for work allocations in which each CPU is responsible for blocks of consecutive elements.

Example 8-8 shows the above vector operation parallelized for SN0.

Example 8-8 Fortran Vector Operation, Parallelized

```
      integer i, j, n, niters
      parameter (n = 8*1024*1024, niters = 1000)
      real a(n), b(n), q
c initialization
c$doacross local(i), shared(a,b)
      do i = 1, n
         a(i) = 1.0 - 0.5*i
         b(i) = -10.0 + 0.01*(i*i)
      enddo
c real work
      do it = 1, niters
         q = 0.01*it
c$doacross local(i), shared(a,b,q)
         do i = 1, n
            a(i) = a(i) + q*b(i)
         enddo
      enddo
```

Because the schedule type is not specified, it defaults to simple: that is., process 0 performs iterations 1 to n/p, process 1 performs iterations $1 + (n/p)$ to $2 \times n/p$, and so on. Each process accesses blocks of memory with stride 1.

Because the initialization takes a small amount of time compared with the "real work," parallelizing it doesn't reduce the sequential time of this code by much. Some programmers wouldn't bother to parallelize the first loop for a traditional shared memory computer. However, if you are relying on the first-touch policy to ensure a good data placement, it is critical to parallelize the initialization code in exactly the same way as the processing loop.

Due to the correspondence of iteration number with data element, the parallelization of the "real work" loop means that elements 1 to n/*p* of the vectors *a* and *b* are accessed by process 0. To minimize memory access times, you would like these data elements to be allocated from the memory of the node running process 0. Similarly, elements 1 + (n/*p)* to 2 × n/*p* are accessed by process 1, and you would like them allocated from the memory of the node running it, and so on. This is accomplished automatically by the first-touch policy. The CPU that first touches a data element causes the page holding that data element to be allocated from its local memory. Thus, if the data are to be allocated so that each CPU can make local accesses during the "real work" section of the code, each CPU must be the one to initialize its share of the data. This means that the initialization loop must be parallelized the same way as the "real work" loop.

Now consider why the simple schedule type is important to Example 8-8. Data are placed in units of one page, so they will end up in their optimal location only if the same CPU processes all the data elements in a page. The default page size is 16 KB, or 4096 `REAL*4` data elements; this is a fairly large number. Since the simple schedule type blocks together as many elements as possible for a single CPU to work on (n/*p*), it will create more optimally-allocated pages than any other work distribution.

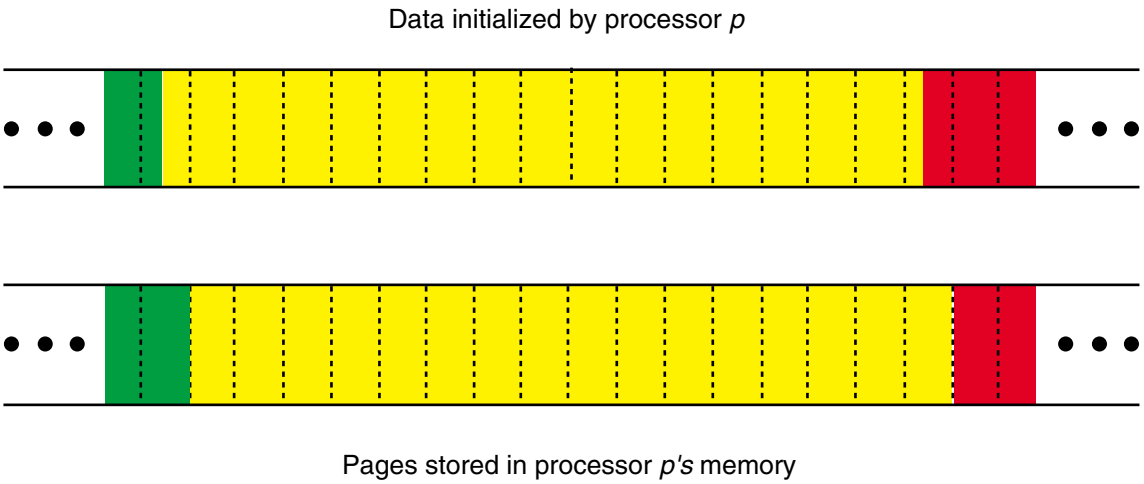


Figure 8-7 Effect of Page Granularity in First-Touch Allocation

For Example 8-8, n = 8 × 1024 × 1024 = 8388608. If the program is run on 128 CPUs, n/p = 65536, which means that each CPU's share of each array fills sixteen 16 KB pages ((65536 elements × 4) ÷ 16 KB). However, it is unlikely that any array begins exactly on a page boundary, so you would expect 15 of a CPU's 16 pages to contain only elements it processes, and one page to contain some of its elements and some of another CPU's elements. This effect is diagrammed in Figure 8-7.

In a perfect data layout, no pages would share elements from multiple CPUs, but this small imperfection has a negligible effect on performance.

On the other hand, if you used an interleaved schedule type, all 128 CPUs repeatedly attempt to concurrently touch 128 adjacent data elements. Because 128 adjacent data elements are almost always on the same page, the resulting data placement could be anything from a random distribution of the pages, to one in which all pages end up in the memory of a single CPU. This initial data placement will certainly affect the performance.

In general, you should try to arrange it so that each CPU's share of a data structure exceeds the size of a page. For finer granularity, you need the directives discussed under "Using Reshaped Distribution Directives" on page 232.

First-Touch Placement with Multiple Data Distributions

Programming for the default first-touch policy is effective if the application requires only one distribution of data. When different data placements are required at different points in the program, more sophisticated techniques are required. The data directives (see "Using Data Distribution Directives" on page 222) allow data structures to be redistributed at execution time, so they provide the most general approach to handling multiple data placements.

In many cases, you don't need to go to this extra effort. For example, the initial data placement can be optimized with first-touch, and migration can redistribute data during the run of the program. In other cases, copying can be used to handle multiple data placements. As an example, consider the FT kernel of the NAS FT benchmark (see the link under "Third-Party Resources" on page xxx). This is a three-dimensional Fast Fourier Transform (FFT) carried out multiple times. It is implemented by performing one-dimensional FFTs in each of the three dimensions. As long as you don't need more than 128-way parallelism, the x- and y-calculations can be parallelized by partitioning data along the z-axis, as sketched in Figure 8-8.

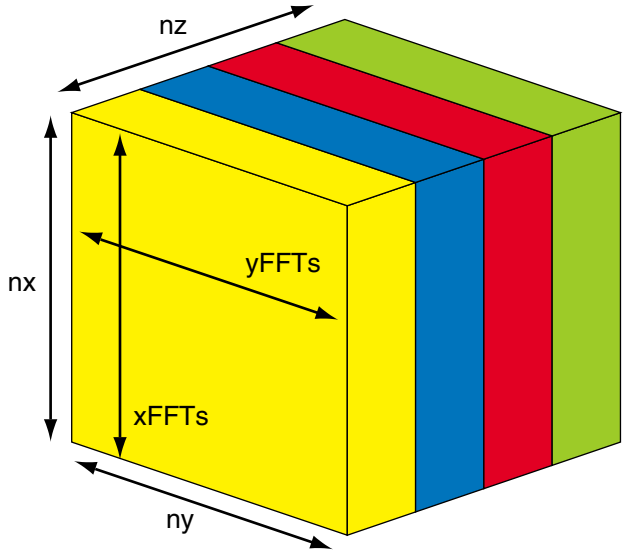


Figure 8-8 Data Partition for NAS FT Kernel

For SN0, first-touch is used to place nz/p, xy-planes in the local memory of each CPU. The CPUs are then responsible for performing the FFTs in their shares of planes.

Once the FFTs are complete, z-transforms need to be done. On a conventional distributed memory computer, this presents a problem. The xy-planes have been distributed, no CPU has access to all the data along the lines in the z-dimension, so the z-FFTs cannot be performed. The conventional solution is to redistribute the data by transposing the array, so that each CPU holds a set of zy-planes. One-dimensional FFTs can then be carried out along the first dimension in each plane. A transpose is convenient since it is a well-defined operation that can be optimized and packaged into a library routine. In addition, it moves a lot of data together, so it is more efficient than moving individual elements to perform z-FFTs one at a time. (This technique was discussed earlier at “Understanding Transpositions” on page 153.)

SN0, however, is a shared memory computer, so explicit data redistribution is not needed. Instead, split up the z-FFTs among the CPUs by partitioning work along the y-axis, as sketched in Figure 8-9.

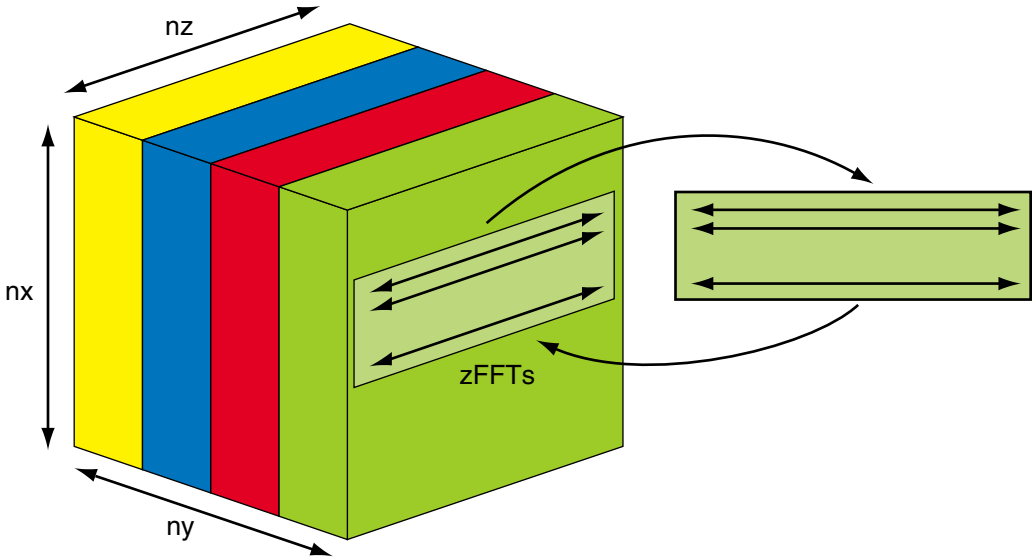


Figure 8-9 NAS FT Kernel Data Redistributed

Each CPU then copies several z-lines at a time into a scratch array, performs the z-FFTs on the copied data, and completes the operation by copying back the results.

Copying has several advantages over explicitly redistributing the data:

- It brings several z-lines directly into each CPU's cache, and the z-FFTs are performed on these in-cache data. A transpose moves nonlocal memory to local memory, which must then be moved into the cache in a separate step.
- Copying reduces TLB misses. The same technique was recommended earlier under "Using Copying to Circumvent TLB Thrashing" on page 146.
- Copying scales well because it is perfectly parallel. Parallel transpose algorithms require frequent synchronization points.
- Copy is easy to implement. An optimal transpose is not simple code to write.

Combining first-touch placement with copying solves a problem that might otherwise require two data distributions.

Using Data Distribution Directives

You can create almost any data placement using the first-touch policy. However, it may not be easy for others to see what you are trying to accomplish. Furthermore, changing the data placement could require modifying a lot of code. An alternate way to distribute data is through compiler directives. Directives state explicitly how the data are distributed. Modifying the data distribution is easy; it only requires updating the directives, rather than changing the program logic or the way the data structures are defined.

Understanding Directive Syntax

The MIPSpro 7.2 compilers support two types of data distribution directives: regular and reshaping. Both allow you to place blocks or stripes of arrays in the memory of the CPUs that operate on those data. The regular data distribution directives are limited to units of whole virtual pages, whereas the reshaping directives permit arbitrary granularity.

Data distribution directives are supported for both Fortran and C. For Fortran, two forms of the directives are accepted: an old form starting with C$ and a new form that extends the OpenMP directive set. For C and C++, the directives are written as pragmas.

Despite the differences in syntax, there are only five directives, plus an extended clause to the directive for parallel processing. These are supported in all languages beginning with compiler release 7.2. They are summarized in Table 8-1.

Table 8-1 Forms of the Data Distribution Directives

Purpose	C and C++	Old Fortran	OpenMP Fortran
Define regular distribution of an array	#pragma distribute	C$DISTRIBUTE	!$SGI DISTRIBUTE
Define reshaped distribution of an array	#pragma distribute_reshape	C$DISTRIBUTE_RESHAPE	!$SGI DISTRIBUTE_RESHAPE
Permit dynamic redistribution of an array	#pragma dynamic	C$DYNAMIC	!$SGI DYNAMIC
Force redistribution of an array	#pragma redistribute	C$REDISTRIBUTE	!$SGI REDISTRIBUTE
Specify distribution by pages	#pragma page_place	C$PAGE_PLACE	!$SGI PAGE_PLACE
Associate parallel threads with distributed data	#pragma pfor ... affinity(*idx*) = data(*array*(*expr*)) ...	C$DOACROSS ... AFFINITY(*idx*) = data(*array*(*expr*)) ...	!$OMP PARALLEL DO... !$SGI AFFINITY(*idx*) = data(*array*(*expr*)) ...

All directives but the last one have the same verb and arguments in each language; only the fixed text that precedes the verb varies (#pragma, C$, or !$OMP). This book refers to these directives by their verbs: Distribute, Distribute_Reshape, Dynamic, Redistribute, and Page_Place. The final directive is named by its OpenMP verb, Parallel Do.

The examples in the following sections use old Fortran syntax. When writing new Fortran code, use the OpenMP directive syntax.

For detailed documentation of these directives, see the manuals listed in "Compiler Manuals" on page xxix. The manuals in particular are helpful:

- The *Fortran 90 Commands and Directives* manual displays OpenMP directives and has a detailed discussion of their use.
- The *MIPSpro Fortran 77 Programmer's Guide* displays old Fortran syntax, with some discussion of use.

Using Distribute for Loop Parallelization

An example is the easiest way to show the use of the Distribute directive. Example 8-9 shows the easily-parallelized vector operation of Example 8-7 modified with directives to ensure proper placement of the arrays.

Example 8-9 Fortran Vector Operation with Distribution Directives

```
      integer i, j, n, niters
      parameter (n = 8*1024*1024, niters = 1000)
c---Note that the distribute directive FOLLOWS the array declarations.
      real a(n), b(n), q
c$distribute a(block), b(block)
c initialization
      do i = 1, n
         a(i) = 1.0 - 0.5*i
         b(i) = -10.0 + 0.01*(i*i)
      enddo
c real work
      do it = 1, niters
         q = 0.01*it
c$doacross local(i), shared(a,b,q), affinity (i) = data(a(i))
         do i = 1, n
            a(i) = a(i) + q*b(i)
         enddo
      enddo
```

Two directives are used: The first, Distribute, instructs the compiler to allocate the memory for the arrays *a* and *b* from all nodes on which the program runs. The second directive, Parallel Do, uses the clause AFFINITY (I) = DATA (A(I)), to tell the compiler to distribute work to the CPUs based on how the contents of array *a* are distributed.

Note that because the data is explicitly distributed, it is no longer necessary to parallelize the initialization loop to get correct first-touch allocation (although it is still good programming practice to parallelize data initializations).

Using the Distribute Directive

In Example 8-9, Distribute specifies a BLOCK mapping for both arrays. BLOCK means that, when running with *p* CPUs, each array is to be divided into *p* blocks of size ceiling(n/*p*), with the first block assigned to the first CPU, the second block assigned to the second CPU, and so on. The intent of assigning blocks to CPUs is to allow each block to be stored in one CPUs' local memory.

Only whole pages are distributed, so when a page straddles blocks belonging to different CPUs, it is stored entirely in the memory of a single node. As a result, some CPUs will use nonlocal accesses to some of the data on one or two pages per array. (The situation is diagrammed in Figure 8-7 on page 218.) An imperfect data distribution has a negligible effect on performance because, as long as a "block" comprises at least a few pages, the ratio of nonlocal to local accesses is small. When the block size is less than a page, you must live with a larger fraction of nonlocal accesses, or use the reshaped directives. (Data placement is rarely important for arrays smaller than a page, because if they are used heavily, they fit entirely in cache.)

Using Parallel Do with Distributed Data

Now look at the Parallel Do directive (C$DOACROSS) in Example 8-9. It instructs the compiler to run this loop in parallel. But instead of distributing iterations to CPUs by specifying a schedule type such as "simple" or "interleave," the AFFINITY clause is used. This new clause tells the compiler to execute iteration *i* on the CPU that is responsible for data element *a(i)* under data distribution. Thus work is divided among the CPUs according to their access to local data. This is the situation achieved using first-touch allocation and default scheduling, but now it is specified explicitly.

Page granularity is not considered in assigning work; the first CPU carries out iterations 1 to ceil(n/p), the second CPU performs iterations ceil(n/p) + 1 to 2 × ceil(n/p), and so on. This ensures a proper balance of work among CPUs, at the possible expense of a few nonlocal memory accesses.

For the default BLOCK distribution used in this example, an Affinity clause assigns work to CPUs identically to the default simple schedule type. The Affinity clause, however, has advantages. First, for some complicated data distributions, there are no defined schedule types that can achieve optimal work distribution. Second, the Affinity clause makes the code easier to maintain, because you can change the data distribution without having to modify the Distribute directive to realign the work with the data.

Understanding Distribution Mapping Options

The Distribute directive allows you specify a different mapping for each dimension of each distributed array. The possible mappings are these:

- An asterisk, to indicate you don't specify a distribution for that dimension.
- BLOCK, meaning one block of adjacent elements is assigned to each CPU.
- CYCLIC, meaning sequential elements on that dimension are dealt like cards to CPUs in rotation. (In poker and bridge, cards are dealt to players in CYCLIC order.)
- CYCLIC(x), meaning blocks of x elements on that dimension are dealt to CPUs in rotation. (In the game of pinochle, cards are customarily dealt CYCLIC(3).)

For example, suppose a Fortran program has a two-dimensional array A and uses p CPUs. Then C$DISTRIBUTE A(*,CYCLIC) assigns columns 1, p+1, 2p+1, ... of A to the first CPU; columns 2, p+2, 2p+2, ... to the second CPU; and so on. CYCLIC(x) specifies a block-cyclic mapping, in which blocks, rather than individual elements, are cyclically assigned to CPUs, with the block size given by the compile-time expression x. For example, C$DISTRIBUTE A(*,CYCLIC(2)) assigns columns 1, 2, 2p+1, 2p+2, ... to the first CPU; 3, 4, 2p+3, 2p+4, ... to the second CPU; and so on.

Combinations of the mappings on different dimensions can produce a variety of distributions, as shown in Figure 8-10, in which each color represents the intended assignment of data elements to one CPU.

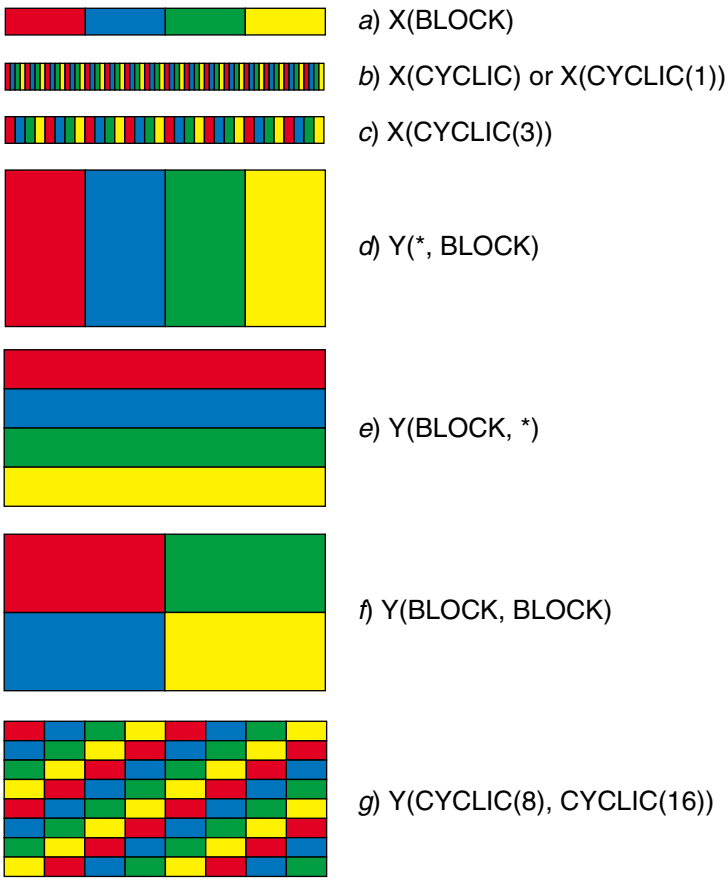


Figure 8-10 Some Possible Regular Distributions

The distributions illustrated in Figure 8-10 are ideals, and do not take into consideration the restriction to whole pages. The intended distribution of data is achieved only when at least a page of data is assigned to a CPU's local memory. In the figure, mappings (a) and (d) produce the desired data distributions for arrays of moderate size, while mappings (e) and (f) require large arrays for the data placements to be near-optimal. For the cyclic mappings (b, c, and g) reshaped data distribution directives to achieve the intended results ("Using Reshaped Distribution Directives" on page 232).

Understanding the ONTO Clause

When an array is distributed in multiple dimensions, data is apportioned to CPUs as equally as possible across each dimension. For example, if an array has two distributed dimensions, execution on six CPUs assigns the first dimension to three CPUs and the second to two (3 x 2 = 6). The optional ONTO clause allows you to override this default and explicitly control the number of CPUs in each dimension. The clause ONTO(2,3) assigns two CPUs to the first dimension and three to the second. Some possible arrangements are sketched in Figure 8-11.

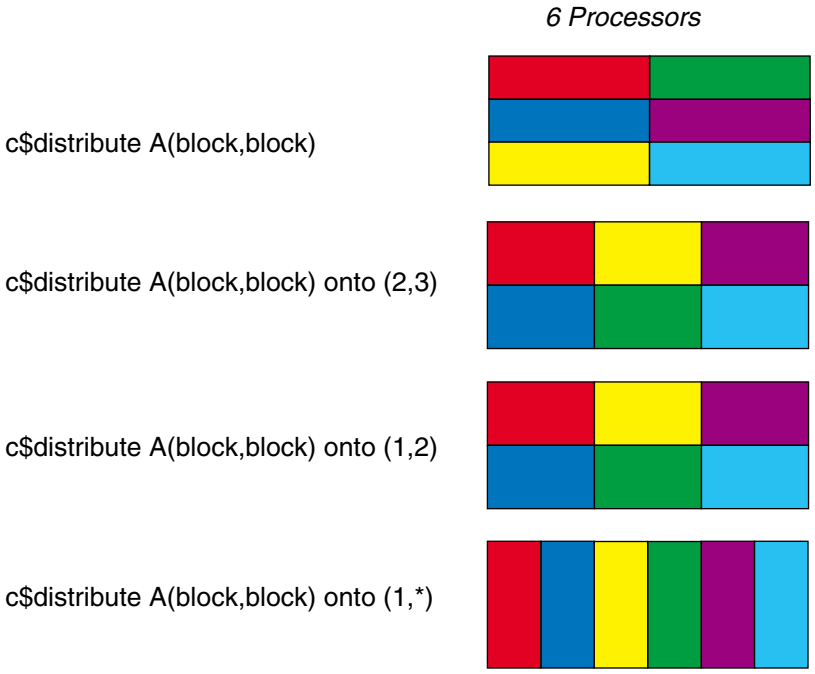


Figure 8-11 Possible Outcomes of Distribute ONTO Clause

The arguments of ONTO specify the aspect ratio desired. If the available CPUs cannot exactly match the specified aspect ratio, the best approximation is used. In a six-CPU execution, ONTO(1,2) assigns two CPUs to the first dimension and three to the second. An argument of asterisk means that all remaining CPUs are to fill in that dimension: ONTO(2,*) assigns two CPUs to the first dimension and $p/2$ to the second.

Understanding the AFFINITY Clause for Data

The Affinity clause extends the Parallel Do directive to data distribution. Affinity has two forms. The data form, in which iterations are assigned to CPUs to match a data distribution, is used in Example 8-9 on page 223. In that example, the correspondence between iterations and data distribution is quite simple. The clause, however, allows more complicated associations. One is shown in Example 8-10.

Example 8-10 Parallel Loop with Affinity in Data

```
c$distribute a(block,cyclic(1))
c$doacross local(i,j), shared(a)
c$&,       affinity(i) = data(a(2*i+3,j))
      do i = 1, n
         do j = 1, n
            a(2*i+3,j) = a(2*i+3,j-1)
         enddo
      enddo
```

Here the compiler and runtime try to execute loop iterations on *i* in those CPUs for which element `DATA(2*i+3,j)` is in local memory. The loop-index variable (*i* in the example) cannot appear in more that one dimension of the clause, and the expressions involving it are limited to the form $a \times i + b$, where a and b are literal constants with a greater than zero.

Understanding the AFFINITY Clause for Threads

A different kind of affinity relates iterations to threads of execution, without regard to data location. This form of the clause is shown in Example 8-11.

Example 8-11 Parallel Loop with Affinity in Threads

```
      integer n, p, i
      parameter (n = 8*1024*1024)
      real a(n)
      p = 1
c$    p = mp_numthreads()
c$doacross local (i), shared (a,p)
c&,        affinity(i) = thread(i/((n+p-1)/p))
      do i = 1, n
         a(i) = 0.0
      enddo
```

This form of affinity has no direct relationship to data distribution. Rather, it replaces the schedule type (that is, #pfor schedtype, or !$OMP DO SCHEDULE) with a schedule based on an expression in an index variable. The code in Example 8-11 executes iteration *i* on the thread given by an expression, modulo the number of threads that exist. The expression may need to be evaluated in each iteration of the loop, so variables (other than the loop index) must be declared shared and not changed during the execution of the loop.

Understanding the NEST Clause

Multi-dimensional mappings (for example, mappings f and g in Figure 8-10 on page 226) often benefit from parallelization over more than just the one loop to which a standard directive applies. The NEST clause permits such parallelizations. It specifies that the full set of iterations in the loop nest may be executed concurrently. Typical syntax is shown in Example 8-12.

Example 8-12 Loop Parallelized with the NEST Clause

```
      real a(n,n)
c$doacross nest(i,j), local(i,j), shared(a)
      do j = 1, n
         do i = 1, n
            a(i,j) = 0.0
         enddo
      enddo
```

The normal and default parallelization of the loop in Example 8-12 would schedule the iterations of each column of the array—iterations in *i* for each value of *j*—on parallel CPUs. The work of the outer loop, which is only index incrementing, is done serially. In the example, the NEST clause specifies that all $n \times n$ iterations can be executed in parallel. To use this directive, the loops to be parallelized must be perfectly nested, that is, no code is allowed except within the innermost loop. Using NEST, when there are enough CPUs, every combination of *i* and *j* executes in parallel. In any event, when there are more than *n* CPUs, all will have work to do.

Example 8-13 Loop Parallelized with NEST Clause with Data Affinity

```
real a(n,n)
c$distribute a(block,block)
c$doacross nest(i,j), local(i,j), shared (a,n)
c$&,       affinity(i,j) = data(a(i,j))
      do j = 1, n
         do i = 1, n
            a(i,j) = 0.0
         enddo
      enddo
```

Example 8-13 shows the combination of the NEST and AFFINITY (to data) clauses. All iterations of the loop are treated independently, but each CPU operates on its share of distributed data.

Example 8-14 Loop Parallelized with NEST, AFFINITY, and ONTO

```
real a(m,n)
c$distribute a(block,block)
c$doacross nest(i,j), local(i,j), shared (a,m,n)
c$&,       affinity(i,j) = data(a(i,j)) onto (2,1)
      do j = 1, n
         do i = 1, m
            a(i,j) = 0.0
         enddo
      enddo
```

Example 8-14 shows the combination of all three clauses: NEST to parallelize all iterations; AFFINITY to causes CPUs to work on local data; and an ONTO clause to specify the aspect ratio of the parallelization. In this example, twice as many CPUs are used to parallelize the *i*-dimension as the *j*-dimension.

Understanding the Redistribution Directives

The Distribute directive is commonly used to specify a single arrangement of data that is constant throughout the program. In some programs you need to distribute an array differently during different phases of the program. Two directives allow you to accomplish this easily:

- The Dynamic directive tells the compiler that an array may need to be redistributed, so its distribution must be calculated at runtime.
- The Redistribute directive causes an array to be distributed differently.

Redistribute allows you to change the distribution defined by a Distribute directive to another mapping. However, redistribution should not be seen as a general performance tool. For example, the FFT algorithm is better with data copying, as discussed under "First-Touch Placement with Multiple Data Distributions" on page 219.

(Despite its name, the Distribute_Reshape directive does not perform redistribution. It tells the compiler that it can reshape the distributed data into nonstandard layouts.)

Using the Page_Place Directive for Custom Mappings

The block and cyclic mappings are well-suited for regular data structures such as numeric arrays. The Page_Place directive can be used to distribute nonrectangular data. This directive allows you to assign ranges of whole pages to precisely the CPUs you want. The penalty is that you have to understand the memory layout of your data in terms of virtual pages.

The Page_Place directive, unlike most other directives, is an executable statement (not an instruction to the compiler on how to treat the program). It takes three arguments:

- The name of a variable, whose first address is the start of the area to be placed
- The size of the area to be placed
- The number of the CPU, where the first CPU being used for this program is 0

Because the directive is executable, you can use it to place dynamically allocated data, and you can use it to move data from one memory to another. (The Distribute directive is not an executable statement, so it can reference only the data that is statically declared in the code.) In Example 8-15, the Page_Place directive is used to create a block mapping of the array *a* onto *p* CPUs.

Example 8-15 Fortran Code for Explicit Page Placement

```
      integer n, p, npp, i
      parameter (n = 8*1024*1024)
      real a(n)
      p = 1
c$    p = mp_numthreads()
c-----distribute a using a block mapping
      npp = (n + p-1)/p       ! number of elements per CPU
c$doacross local(i), shared(a,npp)
      do i = 0, p-1
c$page_place (a(1 + i*npp), npp*4, i)
      enddo
```

If the array has not previously been distributed via first-touch or a Distribute directive, the Page_Place directive establishes the initial placement and incurs no time cost. If the data have already been placed, execution of the directive redistributes the data, causing the pages to migrate from their initial locations to those specified by the directive. This data movement requires some time to complete. (It may or may not be faster than data copying.) If your intention is only to set the initial placement, rather than redistributing the pages, make the directive the first executable statement that acts on the specified data, and don't use other distribution directives on the same variables.

When your intention is to redistribute the data, Page-Place will accomplish this. But for regularly distributed data, the Redistribute directive also works and is more convenient.

Using Reshaped Distribution Directives

The regular data distribution directives provide an easy way to specify the desired data distribution, but they have a limitation: the distributions operate only on whole virtual pages. This limits their use with small arrays and cyclic distributions. For these situations another directive is provided, Distribute_Reshape.

The arguments to Distribute_Reshape are the same as those of the Distribute directive, and it has the same basic purpose. However, it also promises the compiler that the program makes no assumptions about the layout of data in memory; for example, that it does not assume that array elements with consecutive indexes are located in consecutive virtual addresses. With this assurance, the compiler can distribute data in ways that are not possible when it has to preserve the standard assumptions about the layout of data in memory. For example, when the size of a distributed block is less than a page, the compiler can pad each block of elements to a full page and distribute that. The implications of this are best seen through an example.

Example 8-16 shows the declaration of a three-dimensional array, *a*, and distributes its xy-planes to CPU local memory using a reshaped cyclic mapping.

Example 8-16 Declarations Using the Distribute_Reshape Directive

```
      integer i, j, k
      integer n1, n2, n3
      parameter (n1=200, n2=150, n3=150)
      real a(n1, n2, n3)
c$distribute_reshape a(*,*,cyclic)
      real sum
```

The reshaped data distribution means that each CPU is assigned ceil($n3/p$) xy-planes in a cyclic order. The x- and y-dimensions are not distributed, so each plane consumes n1 × n2 consecutive data locations in normal Fortran storage order (that is, the first *n1* elements are down the first column, the next *n1* elements are down the second column, and so on), and these are in the expected, consecutive, virtual storage locations.

However, the Distribute_Reshape directive promises the compiler that the program makes no assumptions about the location of one z-plane compared with another. That is, even though standard Fortran storage order specifies that `a(1,1,2)` is stored n1 × n2 elements after `a(1,1,1)`, this is no longer guaranteed when the reshape directive is used. The compiler is allowed to insert "holes," or padding space, between these planes as needed in order to perform the exact, cyclic distribution requested. The situation is reflected in Figure 8-12.

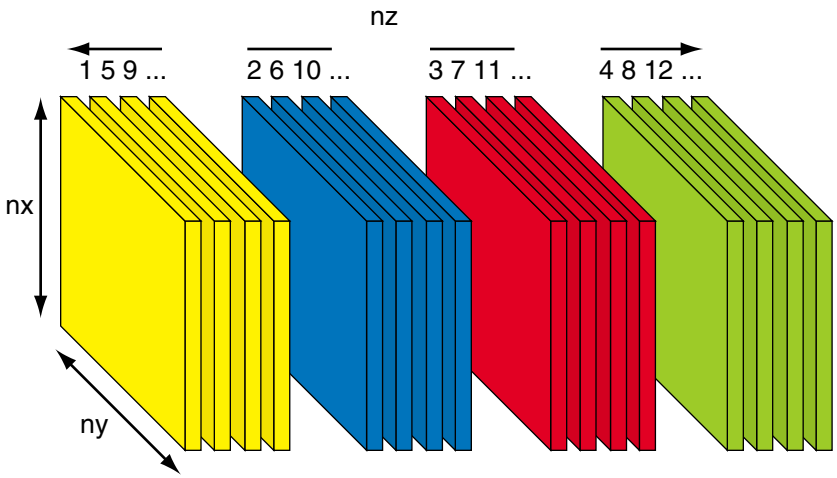


Figure 8-12 Reshaped Distribution of Three-Dimensional Array

In Example 8-17, the BLAS-1 routines **sasum** and **sscal** from the CHALLENGEcomplib are applied in parallel to pencils of data in each of the three dimensions. The library routine **sasum** knows nothing about reshaped arrays. On the contrary, it assumes that it is working on a vector of consecutive reals. Its first argument tells it how many elements to sum; the second is the address at which to start summing; and the third argument is the stride, that is, the number of storage elements by which to increment the address to get to the next value included in the sum.

Example 8-17 Valid and Invalid Use of Reshaped Array

```
c$doacross local(k,j,sum), shared(a)
      do k = 1, n3
         do j = 1, n2
            sum = sasum(n1, a(1,j,k), 1)
            call sscal(n1, sum, a(1,j,k), 1)
         enddo
      enddo
c$doacross local(k,i,sum), shared(a)
      do k = 1, n3
         do i = 1, n1
            sum = sasum(n2, a(i,1,k), n1)
            call sscal(n2, sum, a(i,1,k), n1)
         enddo
      enddo
c$doacross local(j,i,sum), shared(a)
      do j = 1, n2
         do i = 1, n1
            sum = sasum(n3, a(i,j,1), n1*n2)
            call sscal(n3, sum, a(i,j,1), n1*n2)
         enddo
      enddo
```

In the first loop in Example 8-17, values are summed down the x-dimension. These are adjacent to one another in memory, so the stride is 1. In the second loop, values are summed in the y-dimension. Each y-value is one column of length *n1* away from its predecessor, so these have a stride of *n1*. The x- and y-dimensions are not distributed, so these values are stored at the indicated strides, and the subroutine calls generate the expected results.

This is not the case for the z-dimension . You can make no assumptions about the storage layout in a distributed dimension. The stride argument of $n1 \times n2$ used in the third loop, while logically correct, cannot be assumed to describe the storage addressing. Depending on the size and alignment of the array, the compiler may have inserted padding between elements to achieve perfect distribution. This third loop generates incorrect results when run in parallel. The **sasum** routine adds the value of some padding in place of some of the data elements. (If it is compiled without the *-mp* flag so that the directives are ignored, it does generate correct results.)

The problem is the result of using a library routine that makes assumptions about the storage layout. To correct this you have to remove those assumptions. One solution is to use standard array indexing instead of the library routine. Example 8-18 replaces the library call with code that does not assume a storage layout.

Example 8-18 Corrected Use of Reshaped Array

```
c$doacross local(j,i,k,sum), shared(a)
      do j = 1, n2
         do i = 1, n1
            sum = 0.0
            do k = 1, n3
               sum = sum + abs(a(i,j,k))
            enddo
            do k = 1, n3
               a(i,j,k) = a(i,j,k)/sum
            enddo
          enddo
      enddo
```

A solution that allows you to employ library routines is to use copying. Example 8-19 shows this approach.

Example 8-19 Gathering Reshaped Data with Copying

```
c$doacross local(j,i,k,sum,tmp), shared(a)
      do j = 1, n2
         do i = 1, n1
            do k = 1, n3
               tmp(k) = a(i,j,k)
            enddo
            sum = sasum(n3, tmp, 1)
            call sscal(n3, sum, tmp, 1)
         enddo
      enddo
```

The copy loop accesses array elements by their indexes, so it does not make assumptions about the storage layout—the compiler generates code that allows for any reshaping it has done. When a local scratch array is used, the stride is known, so **sasum** may be used safely. Note, though, that the copy step in Example 8-19 accesses memory in long strides. The version in Example 8-20 involves more code and scratch space, but it makes much more effective use of cache lines.

Example 8-20 Gathering Reshaped Data with Cache-Friendly Copying

```
c$doacross local(j,i,k,sum,tmp), shared(a)
      do j = 1, n2
         do k = 1, n3
            do i = 1, n1
               tmp(i,k) = a(i,j,k)
            enddo
         enddo
         do i = 1, n1
            sum = sasum(n3, tmp(i,1), n1)
            call sscal(n3, sum, tmp(i,1), n1)
         enddo
      enddo
```

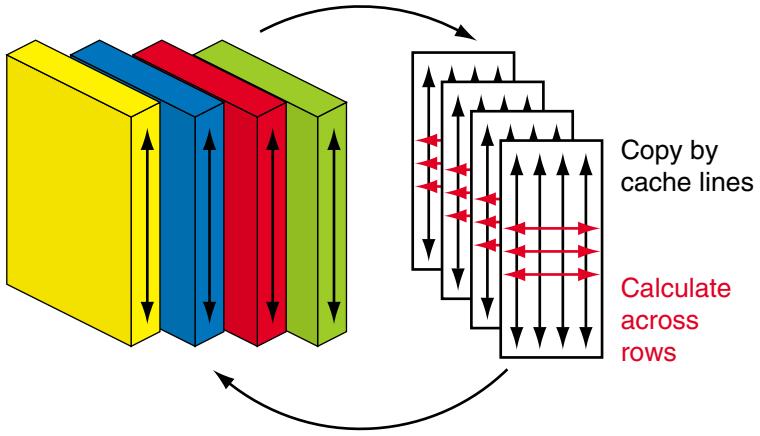


Figure 8-13 Copying By Cache Lines for Summation

Distribute_Reshape, although implemented for shared memory programs, is essentially a distributed memory construct. You use it to partition data structures the way you would on a distributed-memory computer. In a true distributed-memory system, once the data structures are partitioned, accessing them in the distributed dimension is restricted. In considering whether you can safely perform an operation on a reshaped array, ask if it is something that wouldn't be allowed on a distributed-memory computer. However, if the operation makes no assumptions about data layout, SN0 shared memory allows you to conveniently perform many operations that would be much more difficult to implement on a distributed memory computer.

Creating Your Own Reshaped Distribution

The purpose of the Distribute_Reshape directive is to achieve the desired data distribution without page-granularity limitations. You can accomplish the same thing through the use of dynamic memory allocation and pointers. The method follows this outline:

1. Allocate enough space for each CPU so that it doesn't need to share a page with another CPU.
2. Pack the pieces each CPU is responsible for into its pages of memory.
3. Use first-touch allocation, or Page_Place, to make sure each CPU's pages are stored locally.
4. Set up an array of pointers so the individual pieces can be accessed conveniently.

However, this approach can easily add great complexity to what ought to be a readable algorithm. As with most optimizations, it is usually better to leave the source code simple and let the compiler do the hard work behind the scenes.

Restrictions of Reshaped Distribution

When you use the Distribute_Reshape directive, there are some restrictions you should remember:

- The distribution of a reshaped array cannot be changed dynamically (that is, there is no redistribute_reshape directive).
- Static-initialized data cannot be reshaped.
- Arrays that are explicitly allocated through alloca() or malloc() and accessed though pointers cannot be reshaped.
- An array that is equivalenced to another array cannot be reshaped.
- I/O for a reshaped array cannot be mixed with namelist I/O or a function call in the same I/O statement.
- A COMMON block containing a reshaped array cannot be linked -Xlocal.

In addition, some care must be used in passing reshaped arrays to subroutines. As long as you pass the entire array, and the declared size of the array in the subroutine matches the size of array that was passed to it, problems are unlikely. Example 8-21 illustrates this.

Example 8-21 Reshaped Array as Actual Parameter—Valid

```
      real a(n1,n2)
c$distribute_reshape a(block,block)
      call sub(a,n1,n2)
                . . .
      subroutine sub(a,n1,n2)
      real a(n1,n2)
                . . .
```

In Example 8-22, the formal parameter is declared differently from the actual reshaped array. Treating the two-dimensional array as a vector amounts to an assumption that consecutive elements are adjacent in memory, a false assumption.

Example 8-22 Reshaped Array as Actual Parameter—Invalid

```
      real a(n1,n2)
c$distribute_reshape a(block,block)
      call sub(a,n1*n2)
                . . .
      subroutine sub(a,n)
      real a(n)
                . . .
```

Inside a subroutine, you don't need to declare how an array has been distributed. In fact, the subroutine is more general if you do not declare the distribution. The compiler will generate versions of the subroutine necessary to handle all distributions that are passed to it. Example 8-23 shows a subroutine receiving two arrays that are reshaped differently.

Example 8-23 Differently Reshaped Arrays as Actual Parameters

```
      real a(n1,n2)
c$distribute_reshape a(block,block)
      real b(n1,n2)
c$distribute_reshape b(cyclic,cyclic)
      call sub(a,n1,n2)
      call sub(b,n1,n2)
                . . .
      subroutine sub(a,n1,n2)
      real a(n1,n2)
                . . .
```

The compiler actually generates code to handle each distribution passed to the subroutine. As long as the data calculations are efficient for both distributions, this will achieve good performance. If a particular algorithm works only for a specific data distribution, you can declare the required distribution inside the subroutine by using a Distribute_Reshape directive there. Then, calls to the subroutine that pass a mismatched distribution cause compile- or link-time errors.

Most errors in accessing reshaped arrays are caught at compile time or link time. However, some errors, such as passing a portion of a reshaped array that spans a distributed dimension, can be caught only at run time. You can instruct the compiler to do runtime checks for these with the *-MP:check_reshape=on* flag. You should use this flag during the development and debugging of programs which use reshaped arrays. In addition, the Fortran manuals listed in "Compiler Manuals" on page xxix have more information about reshaped arrays.

Investigating Data Distributions

Sometimes it is useful to check how data have been distributed. You can do this on a program basis with an environment variable and dynamically with a function call.

Using _DSM_VERBOSE

If you are using the data distribution directives, you can set the environment variable _DSM_VERBOSE. When it is set, the program will print out messages at run time that tell you whether the distribution directives are being used, which physical CPUs the threads are running on, and what the page sizes are. A report might look like Example 8-24.

Example 8-24 Typical Output of _DSM_VERBOSE

```
% setenv _DSM_VERBOSE
% a.out
[ 0]: 16 CPUs, using 4 threads.
        1 CPUs per memory
        Migration: OFF
        MACHINE: IP27 --- is NUMA ---
        MACHINE: IP27 --- is NUMA ---
Created 4 MLDs
Created and placed MLD-set. Topology-free, Advisory.
```

```
MLD attachments are:
        MLD  0. Node /hw/module/3/slot/n4/node
        MLD  1. Node /hw/module/3/slot/n3/node
        MLD  2. Node /hw/module/3/slot/n1/node
        MLD  3. Node /hw/module/3/slot/n2/node
[ 0]: process_mldlink: thread  0 (pid  3832) to memory  0. -advisory-.
Pagesize: stack 16384 data 16384 text 16384 (bytes)
--- finished MLD initialization ---
                                  . . .
[ 0]: process_mldlink: thread  1 (pid  3797) to memory  1. -advisory-.
[ 0]: process_mldlink: thread  2 (pid  3828) to memory  2. -advisory-.
[ 0]: process_mldlink: thread  3 (pid  3843) to memory  3. -advisory-.
[ 0]: process_mldlink: thread  0 (pid  3832) to memory  0. -advisory-.
```

These messages are useful in verifying that the requested data placements are actually being performed.

The utility *dplace* (see "Non-MP Library Programs and Dplace" on page 243) disables the data distribution directives. It should not normally be used to run MP library programs. If you do use it, however, it displays messages reminding you that "DSM" is disabled.

Using Dynamic Placement Information

You can obtain information about data placement within the program with the **dsm_home_threadnum()** intrinsic. It takes an address as an argument, and returns the number of the CPU in whose local memory the page containing that address is stored. It is used as follows:

```
integer dsm_home_threadnum
numthread = dsm_home_threadnum(array(i))
```

Two CPUs are connected to each node and they share the same memory, so the function returns the lowest CPU number of the two running on the node with the data.

The numbering of CPUs is relative to the program. You can also determine which absolute physical node a page of memory is stored in. (Note, though, that this number will typically change from run to run.) The information is gotten via the **syssgi()** system call using the SGI_PHYSP command (see the syssgi(2) reference page). The routine **va2pa()** translates a virtual address to a physical address using this system call (see Example C-9 on page 302). You can translate a physical page address to a node number with the following macro:

```
#define ADDR2NODE(A) ((int) (va2pa(A) >> 32))
```

Note that the node number is generally half the lowest numbered CPU on the node, so if you would prefer to use CPU numbers, multiply the return value by two.

To explore this information, a test program was written that allocated two vectors, *a* and *b*, of size 256 KB, initialized them sequentially, and then printed out the locations of their pages. The default first-touch policy for a run using five CPUs resulted in the page placements shown in Example 8-25.

Example 8-25 Test Placement Display from First-Touch Allocation

```
Distribution for array "a"
address       byte index  thread  proc
-------------------------------------
0xfffffb4760  0x       0       0    12
0xfffffb8000  0x    38a0       0    12
0xfffffbc000  0x    78a0       0    12
0xfffffc0000  0x    b8a0       0    12
0xfffffc4000  0x    f8a0       0    12
0xfffffc8000  0x   138a0       0    12
0xfffffcc000  0x   178a0       0    12
0xfffffd0000  0x   1b8a0       0    12
0xfffffd4000  0x   1f8a0       0    12
0xfffffd8000  0x   238a0       0    12
0xfffffdc000  0x   278a0       0    12
0xfffffe0000  0x   2b8a0       0    12
0xfffffe4000  0x   2f8a0       0    12
0xfffffe8000  0x   338a0       0    12
0xfffffec000  0x   378a0       0    12
0xffffff0000  0x   3b8a0       0    12
0xffffff4000  0x   3f8a0       0    12
Distribution for array "b"
address       byte index  thread  proc
-------------------------------------
0xffffff6e0f8  0x       0       0    12
0xffffff70000  0x    1f08       0    12
0xffffff74000  0x    5f08       0    12
0xffffff78000  0x    9f08       0    12
0xffffff7c000  0x    df08       0    12
0xffffff80000  0x   11f08       0    12
0xffffff84000  0x   15f08       0    12
0xffffff88000  0x   19f08       0    12
0xffffff8c000  0x   1df08       0    12
0xffffff90000  0x   21f08       0    12
0xffffff94000  0x   25f08       0    12
0xffffff98000  0x   29f08       0    12
```

```
0xfffff9c000  0x   2df08       0    12
0xfffffa0000  0x   31f08       0    12
0xfffffa4000  0x   35f08       0    12
0xfffffa8000  0x   39f08       0    12
0xfffffac000  0x   3df08       0    12
```

The sequential initialization combined with the first-touch policy caused all the pages to be placed in the memory of the first thread which, in this run, was CPU 12, or node 6.

Example 8-26 shows the results from the same code after setting the environment variable _DSM_ROUND_ROBIN to use a round-robin page placement.

Example 8-26 Test Placement Display from Round-Robin Placement

```
Distribution for array "a"
address       byte index  thread  proc
-------------------------------------
0xfffffb4750  0x       0       2     2
0xfffffb8000  0x    38b0       0     0
0xfffffbc000  0x    78b0       4     4
0xfffffc0000  0x    b8b0       2     2
0xfffffc4000  0x    f8b0       0     0
0xfffffc8000  0x   138b0       4     4
0xfffffcc000  0x   178b0       2     2
0xfffffd0000  0x   1b8b0       0     0
0xfffffd4000  0x   1f8b0       4     4
0xfffffd8000  0x   238b0       2     2
0xfffffdc000  0x   278b0       0     0
0xfffffe0000  0x   2b8b0       4     4
0xfffffe4000  0x   2f8b0       2     2
0xfffffe8000  0x   338b0       0     0
0xfffffec000  0x   378b0       4     4
0xffffff0000  0x   3b8b0       2     2
0xffffff4000  0x   3f8b0       0     0
Distribution for array "b"
address       byte index  thread  proc
-------------------------------------
0xfffff6e0e8  0x       0       0     0
0xfffff70000  0x    1f18       4     4
0xfffff74000  0x    5f18       2     2
0xfffff78000  0x    9f18       0     0
0xfffff7c000  0x    df18       4     4
0xfffff80000  0x   11f18       2     2
0xfffff84000  0x   15f18       0     0
0xfffff88000  0x   19f18       4     4
```

```
0xfffff8c000  0x   1df18       2     2
0xfffff90000  0x   21f18       0     0
0xfffff94000  0x   25f18       4     4
0xfffff98000  0x   29f18       2     2
0xfffff9c000  0x   2df18       0     0
0xfffffa0000  0x   31f18       4     4
0xfffffa4000  0x   35f18       2     2
0xfffffa8000  0x   39f18       0     0
0xfffffac000  0x   3df18       4     4
```

Round-robin placement really did spread the data over all three nodes used by this 5-CPU run.

Non-MP Library Programs and Dplace

Programs that do not use the MP library include message passing programs employing MPI, PVM and other libraries, as well as "roll your own parallelism" implemented via **fork()** or **sproc()** (see "Explicit Models of Parallel Computation" on page 194). If such programs have been properly parallelized, but do not scale as expected on SN0, they may require data placement tuning.

If you are developing a new parallel program or are willing to modify the source of your existing programs, you can ensure a good data placement by adhering to the following programming practices which take advantage of the default first-touch data placement policy:

- In a program that starts multiple processes using **fork()**, each process should allocate and initialize its own memory areas. Then the memory used by a process resides in the node where the process runs.
- In a program that starts multiple processes using **sproc()**, do not allocate all memory prior to creating child processes. In the parent process, allocate only memory used by the parent process and memory that is global to all processes. This memory will be located in the node where the parent process runs.

  Each child process should allocate any memory areas that it uses exclusively. Those areas will be located in the node where that process runs.

In general, in any program composed of interacting, independent threads, each thread should be the first to touch any memory used exclusively by that thread. This applies to programs in which a thread is a process, or a POSIX thread, as well as to programs based on MPI and PVM.

For all these programs, data placement can be measure or tuned with the utility *dplace*, which allows you to:

- Change the page size used for text, stack, or data.
- Enable dynamic page migration.
- Specify the topology used by the threads of a parallel program.
- Indicate resource affinities.
- Assign memory ranges to particular nodes.

For the syntax of *dplace*, refer to the dplace(1) reference page. Some *dplace* commands use a placement file, whose syntax is documented in dplace(5). If you are not familiar with *dplace*, it would be a good idea to skim these two reference pages now. In addition, there is a subroutine library interface, documented in the dplace(3) reference page, which not only allows you to accomplish all the same things from within a program, but also allows dynamic control over some of the data placement choices.

Although *dplace* can be used to adjust policies and topology for any program, in practice it should not be used with MP library programs because it disables the data placement specifications made via the MP library compiler directives. For these programs, use the MP library environment variables instead of *dplace*.

Changing the Page Size

In "Using Larger Page Sizes to Reduce TLB Misses" on page 147 we discussed using *dplace* to change the page size:

```
% dplace -data_pagesize 64k -stack_pagesize 64k program arguments...
```

This command runs the specified program after increasing the size of two of the three types of pages—data and stack—from the default 16 KB to 64 KB. The text page size is unchanged. This technique causes the TLB to cover a wider range of virtual data addresses, so reducing TLB thrashing.

In general, TLB thrashing only occurs for pages that store data structures. Dynamically allocated memory space, and global variables such as Fortran common blocks, are stored in data segments; their page size is controlled by *-data_pagesize*. Variables that are local to subroutines are allocated on the stack, so it can also be useful to change *-stack_pagesize* when trying to fix TLB thrashing problems. Increasing the *-text_pagesize* is not generally of benefit for scientific programs, in which time is usually spent in the execution of tight loops over arrays.

There are some restrictions on page sizes. First, the only valid page sizes are 16 KB, 64 KB, 256 KB, 1 MB, 4 MB, and 16 MB. Second, the percentage of system memory allocated to the various possible page sizes is a system configuration parameter that the system administrator must specify. If the system has not been configured to allocate 1 MB pages, it has no effect to request this page size. The page size percentages are set with the *systune* command (see the systune(1) reference page, and the *System Configuration* manual listed under "Software Tool Manuals" on page xxx).

Enabling Page Migration

The concept of dynamic page migration is covered under "Dynamic Page Migration" on page 30. Enabling migration tells the operating system to move pages of memory to the nodes that access them most frequently, so a poor initial placement can be corrected. Recall that page placement is not an issue for single-CPU jobs; migration is only a consideration for parallel programs. Furthermore, migration is only a potential benefit to programs employing a shared memory model; MPI, PVM, and other message-passing programs control data layout explicitly, so automated control of data placement would only get in their way.

The IRIX automatic page migration facility is disabled by default, because page migration is an expensive operation that affects overall system performance. The system administrator can temporarily enable page migration for all programs using the *sn* command, or can enable it permanently by using *systune*. (For more information, see the sn(1) and systune(1) reference pages, the comments in */var/sysgen/mtune/numa*, and the *System Configuration* manual listed under "Software Tool Manuals" on page xxx.)

You can decide to use dynamic page migration for a specific program in one of two ways. If the program uses the MP library, use its environment variables as discussed under "Trying Dynamic Page Migration" on page 209. For other programs, you enable migration using *dplace*:

```
% dplace -migration threshold program arguments...
```

The *threshold* argument is an integer between 0 and 100. A threshold value 0 is special; it turns migration off. Nonzero values adjust the sensitivity of migration, which is based on hardware-maintained access counters that are associated with every 4 KB block of memory. (See the refcnt(5) reference page for a description of these counters and of how the IRIX kernel extends them to virtual pages.)

When migration is enabled, the IRIX kernel surveys the counters periodically. Migration is triggered when the difference between local accesses to a page, and accesses from any one other node, exceeds *threshold*% of the maximum counter value.

When you enable migration, use a conservative threshold, say 90 (which specifies that the count of remote accesses exceeds the count of local accesses by 90% of the counter value). The point is to permit enough migration to correct poor initial data placement, but not to cause so much migration that the expense of moving pages is more than the cost of accessing them remotely. You can experiment with smaller values, but keep in mind that smaller values could keep a page in constant motion. Suppose that a page is being used from three different nodes equally. Moving the page from one of the three nodes to another will not change the distribution of counts.

Specifying the Topology

The *topology* of a program is the shape of the connections between the nodes where the parallel threads of the program are executed. When executing a parallel program, the operating system tries to arrange it so that the processes making up the parallel program run on nearby nodes to minimize the access time to shared data. "Nearby" means minimizing a simple distance metric, the number of router hops needed to get from one node to another. For an 8-node (16-CPU) system, the distance between any two nodes using this metric is as shown in Table 8-2.

Table 8-2 Distance in Router Hops Between Any Nodes in an 8-Node System

Node	0	1	2	3	4	5	6	7
0	0	1	2	2	2	2	3	3
1	1	0	2	2	2	2	3	3
2	2	2	0	1	3	3	2	2
3	2	2	1	0	3	3	2	2
4	2	2	3	3	0	1	2	2
5	2	2	3	3	1	0	2	2
6	3	3	2	2	2	2	0	1
7	3	3	2	2	2	2	1	0

Thus, if the first two threads of a 6-way parallel job are placed on node 3, the second and third threads will be placed on node 2 because, at a distance of only 1, it is the closest node. The final two threads may be placed on any of nodes 0, 1, 6, or 7 because they are all a distance of 2 away.

For most applications, this "cluster" topology is perfect, and you need not worry further about topology issues. For those cases in which it may be of benefit, however, *dplace* does give you a way to override the default and specify other topologies. This is done through a placement file, a text file indicating how many nodes should be used to run a program, what topology should be used, and where the threads should run.

Note: In a *dplace* placement file, a node is called a "memory." You specify a topology of memories, and a specify on which memory a thread should run.

Figure 8-14 shows a simple placement file and its effect on the program:

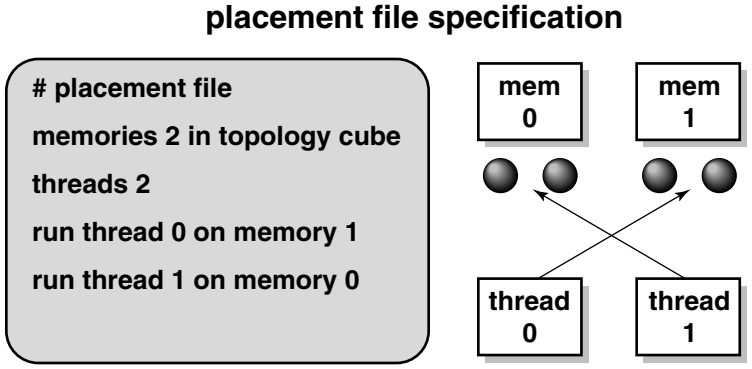


Figure 8-14 Placement File and its Results

With this placement file, two memories are used to run this two-thread program. Without a placement file, IRIX would try to place a two-thread program on a single memory, because two CPUs are associated with each memory, and only one CPU is needed per thread. In addition, the memories are laid out in a cube topology rather than the default cluster topology. For this simple case of two memories, there is no difference; but if more memories had been requested, the cube topology would ensure that memories which should be hypercube neighbors end up adjacent to each other in the SN0 system. Finally, threads are placed onto the memories in the reverse of the typical order, that is, thread 0 is run on memory 1 and thread 1 is run on memory 0.

Placement File Syntax

The syntax of placement files is detailed in the dplace(5) reference page. The key statements in it are the `memories`, `threads`, `run`, and `distribute` statements.

Using Environment Variables in Placement Files

The placement file shown in Figure 8-14 has one limitation: it is not scalable. That is, if you want to change the number of threads, you need to change the placement file. A placement file can refer to environment variables, making it scalable. A file of this type is shown in Example 8-27.

Example 8-27 Scalable Placement File

```
# scalable placement_file
memories $NP in topology cluster # set up memories which are close
threads $NP                      # number of threads
# run the last thread on the first memory etc.
distribute threads $NP-1:0:-1 across memories
```

Instead of writing the number of threads into the file, the environment variable $NP is used. (There is no significance to the name; any variable could be used.) When the placement file is read, the reference to the environment variable is replaced by its value taken from the shell. Thus, if NP is set to 2, the first two statements have the same effect as the version in Figure 8-14. To make the last two statements scalable, they are replaced with a `distribute` statement. This one statement specifies a mapping of all threads to all memories. Its use is covered under "Assigning Threads to Memories" on page 250.

You can use simple arithmetic on environment variables in placement files. A scalable placement file that uses a more conventional placement of two threads per memory is shown in Example 8-28. This example uses the MP_SET_NUMTHREADS environment variable used by the MP library (however, it is not recommended to apply *dplace* to MP library programs). A comparable variable is PT_ITC, the number of processes used by the POSIX Threads library to run a pthreaded program.

Example 8-28 Scalable Placement File for Two Threads Per Memory

```
# Standard 2 thread per memory scalable placement_file
memories ($MP_SET_NUMTHREADS + 1)/2 in topology cluster
threads $MP_SET_NUMTHREADS
distribute threads across memories
```

Using the memories Statement

The `memories` statement specifies the number of memories (SN0 nodes) to be allocated to the program and, optionally, their topology. The topologies supported are these:

- The default of `cube`, specifying a *hypercube* that is a proper subset of the SN0 topology.
- `none`, which is the same as a cluster topology: any arrangement that minimizes router hops between nodes.
- `physical`, used when placing nodes near to hardware, discussed in "Indicating Resource Affinity" on page 251.

The number of nodes in a hypercube is a power of two. As a result, cube topology can have strange side effects when used on machines that do not have a power of two number of nodes. In general, use the cluster topology.

The memories are identified by number, starting from 0, in the order that they are allocated. Using cube or cluster topology, there is no way to relate a memory number to a specific SN0 node; memory 0 could be identified with any node in the system on any run of the program.

Using the threads Statement

The `threads` statement specifies the number of IRIX processes the program will use. The MP library uses IRIX processes as does MPI, PVM, and standard UNIX software. POSIX threads programs are implemented using some number of processes; the number is available within the program using the **pthread_get_concurrency()** function.

The `threads` statement does not create the processes; creation happens as a result of the program's own actions. The `threads` statement tells *dplace* in advance how many threads the program will create. Threads are identified by number, starting from 0, in the sequence in which the program creates them. Thus the first process, the parent process of the program, is thread 0 to *dplace*.

Assigning Threads to Memories

There are two statements used to assign threads to memories, `run` and `distribute`. The `run` statement tells *dplace* to assign a specific thread, by number, to a specific memory by number. Optionally, it can assign the thread to a specific CPU in that node, although there is no particular advantage to doing so. The `run` statement deals with only one thread, so a placement file that uses it cannot be scalable.

The `distribute` statement specifies a mapping of all threads to all memories. In Example 8-27, a `distribute` statement is used to map threads in reverse order to memories, as follows:

```
distribute threads $NP-1:0:-1 across memories
```

The three-part expression, reminiscent of a Fortran DO loop, specifies the threads in the program as *first*:*last*:*stride*. In this example, the last thread (`$NP-1`) is mapped to memory 0, the next-to-last thread is mapped to memory 1, and so on.

This example perversely maps threads to memories in reverse order. If the normal order were desired, the statement could be written in any of the ways shown in Example 8-29.

Example 8-29 Various Ways of Distributing Threads to Memories

```
# Explicitly indicate order of threads
distribute threads 0:$NP-1:1 across memories
# No first:last:stride means 0:last:1
distribute threads across memories
# The stride can be omitted when it is 1
distribute threads 0:$NP-1 across memories
# first:last:stride notation applies to memories too
distribute threads across memories 0:$NP-1:1
distribute threads $NP-1:0:-1 across memories $NP-1:0:-1
```

The `distribute` statement also supports a clause telling how threads are assigned to memories, when there are more threads than memories. The clause can be either block or cyclic. In a block mapping with a block size of two, the first two threads are assigned to the first memory, the second two to the second, and so on. In a cyclic mapping, threads are dealt to memories as you deal a deck of cards. The default is to use a block mapping, and the default block size is ceil(number of threads ÷ number of memories); that is, 2 if there are twice as many threads as memories, and 1 if the number of threads and memories are equal.

Indicating Resource Affinity

The hardware characteristics of an SN0 system are maintained in the hardware graph, a simulated filesystem mounted at */hw*. All real hardware devices and many symbolic pseudo-devices appear in the */hw* filesystem. For example, all the nodes in a system can be listed with the following command:

```
[farewell 1] find /hw -name node -print
/hw/module/5/slot/n1/node
/hw/module/5/slot/n2/node
/hw/module/5/slot/n3/node
/hw/module/5/slot/n4/node
/hw/module/6/slot/n1/node
/hw/module/6/slot/n2/node
/hw/module/6/slot/n3/node
/hw/module/6/slot/n4/node
/hw/module/7/slot/n1/node
/hw/module/7/slot/n2/node
/hw/module/7/slot/n3/node
/hw/module/7/slot/n4/node
/hw/module/8/slot/n1/node
/hw/module/8/slot/n2/node
/hw/module/8/slot/n3/node
/hw/module/8/slot/n4/node
```

This is an Origin2000 system with four modules, 16 nodes, and 32 CPUs.

You can refer to entries in the hardware graph in order to name resources that you want your program to run near. This is done by adding a `near` clause to the `memories` statement in the placement file. For example, if InfiniteReality hardware is installed, it shows up in the hardware graph as follows:

```
% find /hw -name kona -print
/hw/module/1/slot/io5/xwidget/kona
```

In this case, it is in the fifth I/O slot of module 1. You may then indicate that you want your program run near this hardware device by using a line such as the following:

```
memories 3 in topology cluster near /hw/module/1/slot/io5/xwidget/kona
```

You can also use the hardware graph entries and the `physical` topology to run the program on specific nodes:

```
memories 3 in topology physical near /hw/module/2/slot/n1/node
                                     /hw/module/1/slot/n2/node
                                     /hw/module/3/slot/n3/node
```

This line causes the program to be run on node 1 of module 2, node 2 of module 1, and node 3 of module 3. In general, though, it is best to let the operating system pick which nodes to run on, because users' jobs are likely to collide with each other if the physical topology is used.

Assigning Memory Ranges

You can use a placement file to place specific ranges of virtual memory on a particular node. This is done with a `place` statement. For example, a placement file can contain the lines

```
place range 0xfffbffc000 to 0xfffc000000 on memory 0
place range 0xfffd2e4000 to 0xfffda68000 on memory 1
```

Generally, placing address ranges is practical only when used in conjunction with *dprof* (see "Applying dprof" on page 84). This utility profiles the memory accesses in a program. Given the *-pout* option, it writes `place` statements like those shown above to an output file. You can insert these address placement lines into a placement file for the program. Address ranges are likely to change after recompilation, so this use of a placement file is effective only for very fine tuning of an existing binary. Generally it is better to rely on memory placement policies, or source directives, to place memory.

Using the dplace Library for Dynamic Placement

In addition to the capabilities described above, *dplace* can also dynamically move data and threads during program execution. These tasks are accomplished with `migrate` and `move` statements (refer again to the dplace(5) reference page). These statements may be included in a placement file, but they make little sense there. They are meant to be issued at strategic points during the execution of a program. This is done via the subroutine interface described in the dplace(3) reference page.

The subroutine interface is simple; it includes only two functions:

- **dplace_file()** takes as its argument the name of a placement file. It opens, reads, and executes the file, performing multiple statements.
- **dplace_line()** takes a string containing a single *dplace* statement.

Example 8-30 contains some sample code showing how the dynamic specifications can be issued via the library calls.

Example 8-30 Calling dplace Dynamically from Fortran

```
CHARACTER*128 s
np = mp_numthreads()
WRITE(s,*) 'memories ',np,' in topology cluster'
CALL dplace_line(s)
WRITE(s,*) 'threads ',np
CALL dplace_line(s)
DO i=0, np-1
   WRITE(s,*) 'run thread',i,' on memory',i
   CALL dplace_line(s)
   head = %loc( a( 1+i*(n/np) ) )
   tail = %loc( a( (i+1)*(n/np) ) )
   WRITE(s,*) 'place range',head,' to',tail,' on memory',i
   CALL dplace_line(s)
END DO
DO i=0, np-1
   WRITE(s,*) 'move thread',i,' to memory',np-1-i
   CALL dplace_line(s)
END DO
DO i=0, np-1
   head = %loc( a( 1+i*(n/np) ) )
   tail = %loc( a( (i+1)*(n/np) ) )
   WRITE(s,*) 'migrate range',head,' to',tail,' to memory',np-1-i
   CALL dplace_line(s)
END DO
```

The library is linked in with the flag *-ldplace*. The example code calls on **dplace_line()** multiple times to execute the following statements:

- `memories` *p* and `threads` *p*, taking the actual number of processes from the **mp_numthreads** function of the MP library (see the mp(3F) reference page). Note that this code requests a memory (node) per process, which is usually unnecessary.
- A `run` statement for each thread, placing it on the memory of the same number.
- A series of `move` statements to reverse the order of threads on memories.
- A series of `migrate` statements to migrate blocks of array a to the memories of different threads.

Using dplace with MPI 3.1

The current version of the MPI library (3.1 at this writing) is aware of, and compatible with, the MP library. As a result, you rarely need to use dplace with MPI programs. However, in the rare cases when you do want this combination, the key to success is to apply the *mpirun* command to *dplace*, not the reverse. In addition, you must set an environment variable, MPI_DSM_OFF, as in the following two-line example:

```
setenv MPI_DSM_OFF
mpirun -np 4 dplace -place pfile a.out
```

This example starts copies of *dplace* on four CPUs. Each of them, in turn, invokes the program *a.out*, applying the placement file *pfile*.

When using a tool such as *dplace* that produces output on the standard output or standard error streams, it can be difficult to capture the tool's output. The correct approach, as shown in Example 8-31, is to write a small script to do the redirection, and to use *mpirun* to launch the script.

Example 8-31 Using a Script to Capture Redirected Output from an MPI Job

```
> cat myscript
#!/bin/sh
setenv MPI_DSM_OFF
dplace -verbose a.out 2> outfile
> mpirun -np 4 myscript
hello world from process 0
hello world from process 1
hello world from process 2
hello world from process 3
> cat outfile
there are now 1 threads
Setting up policies and initial thread.
Migration is off.
Data placement policy is PlacementDefault.
Creating data PM.
Data pagesize is 16k.
Setting data PM.
Creating stack PM.
Stack pagesize is 16k.
Stack placement policy is PlacementDefault.
Setting stack PM.
there are now 2 threads
there are now 3 threads
there are now 4 threads
there are now 5 threads
```

Advanced Options

The MP library and *dplace* together provide two advanced features that can be of use for realtime programs:

- Running just one process per node
- Locking processes to CPUs

Neither of these features is useful in the typical multi-user environment in which you must share the computer with other users, but they can be crucial for getting minimum response time from a dedicated system.

The first feature can be of use to programs which have high memory bandwidth requirements. Since one CPU can use more than half of a node's bandwidth, running two threads on a node can overload the bandwidth of the hub. Spreading the threads of a job across more nodes can result in higher sustained bandwidth. Of course, you are then limited to half the available CPUs. However, you could run a memory-intensive job with one thread per memory, and concurrently run a cache-friendly (and hence, low-bandwidth) job on the remaining CPUs.

Using the MP library, you can run just one thread per node by setting the environment variable _DSM_PPM to 1. Using *dplace*, write a placement file specifying the same number of memories as threads. In the `distribute` statement, you can explicitly use a block of 1, but this will be the default when the number of threads is the same as the number of memories:

```
memories $NP in topology cluster
threads $NP
distribute theads across memories block 1
```

The second feature, locking threads to CPUs, has long been available in IRIX via the **sysmp**(MP_MUSTRUN,...) system call. The MP library and *dplace* offer convenient access to this capability. For the MP library, you only need to set the _DSM_MUSTRUN environment variable to lock all threads onto the CPUs where they start execution.

Using *dplace*, use the *-mustrun* flag, as follows:

```
% dplace -mustrun -place placement_file program
```

Both the MP library and dplace lock the processes onto the CPUs on which they start. This arrangement will change from run to run. If you want to lock threads onto specific physical CPUs, you must use a placement file specifying a physical topology, or call **sysmp**() explicitly.

Summary

Making a C or Fortran program run in parallel on multiple CPUs can be as simple as recompiling with the *-apo* option, and executing on a system with two or more CPUs. Alternatively, you can write the program for parallel execution using a variety of software models. In every case, the benefit of adding one more CPU depends on the fraction of the code that can be executed in parallel, and on the hardware constraints that can keep the added CPU from pulling its share of the load. You use the profiling tools to locate these bottlenecks, and generally you can relieve them in simple ways, so that the program's performance scales smoothly from one CPU to as many CPUs as you have.

Appendix A

Bentley's Rules Updated

Jon Bentley, a distinguished computer scientist known for writing many publications, published *Writing Efficient Programs*. In this classic (now out of print), Bentley provided a unified, pragmatic treatment of program efficiency, independent of language and host platform.

For ease of presentation, Bentley codified his methods as a set of terse rules—a set of rules that are still useful reminders. This appendix is the list of Bentley's rules, paraphrased from Appendix C of *Writing Efficient Programs* and revised to reflect the needs and opportunities of the SN0 architecture, Silicon Graphics compilers, and the MIPS instruction set. In some cases the rules are only stated as a reminder that the compiler now performs these techniques automatically.

Space-for-Time Rules

Data Structure Augmentation

The time required for common operations on data can often be reduced by augmenting the structure with additional information, or by changing the information within the structure so it can be accessed more easily.

Examples:

- Add a reference counter to facilitate garbage collection.
- Incorporate a "hint" containing a likely, but possibly incorrect, answer.

When the added data can help you avoid a disk access or a memory reference to a different data structure, it is worthwhile. However, when the added hint can at best avoid some inline processing, keep in mind the 100:1 ratio between instruction time and memory-fetch time. It is far more important to minimize secondary cache misses than to minimize instruction counts. If the simple data structure fits in a cache line and the augmented one does not, the time to access the augmented data can be greater than the instructions it saves.

Special applications of this rule include:

- Reorganize data structures to make sure that the most-used fields fit in a single, 128-byte, cache line, and less-used fields fall into separate cache lines. (This of course presumes that each structure is 128-byte aligned in memory.)

  In one example cited in a paper given at Supercomputing '96, most of the cache misses of one program were caused by "a search loop that accesses a four-byte integer field in each element of an array of structures...By storing the integer items in a separate, shadow array, the secondary cache miss rate of the entire application is reduced."

- Reorganize structures so that variables that are updated by different threads are located in different cache lines. The alternative—when variables updated by different threads fall in the same cache line—creates *false sharing*, in which unrelated memory updates force many CPUs to refresh their cache lines.

Store Precomputed Results

The cost of recomputing an expensive function can be reduced by computing the function only once and storing the results. Subsequent requests for the function are handled by table lookup.

Precomputation is almost certainly a win when the replaced computation involves disk access, or an IRIX system call, or even a substantial *libc* function. Precomputation is also a win when it reduces the amount of data the program must reference.

However, keep in mind that if retrieval of the precomputed value leads to an extra cache miss, the replaced computation must have cost at least 100 instructions to break even.

Caching

Data that is accessed most often should be the cheapest to access. However, caching can backfire when locality is not in fact present. Examples:

- The move-to-front heuristic for sequential list management is the classic example.
- In implementing a dictionary, keep the most common words in memory.
- Database systems cache the most-recently used tuples as a matter of course.

The MIPS R10000, SN0 node, and IRIX kernel already cache at a number of levels:

- Primary (on-chip) caches for instructions and data.
- Secondary (on-board) cache for recently-used memory lines of 128 bytes.
- Virtual memory system for most-recently used pages of 16KB and up.
- Filesystems (XFS and NFS independently) for recently-used disk sectors.

(For more detail, see "Understanding the Levels of the Memory Hierarchy" on page 135.) However, all the standard caches are shared by multiple processes. From the standpoint of your program, the other processes are "diluting" the cache with their data, displacing your data. Also, system cache management knows nothing about your algorithms. Thus, application-dependent caching can yield huge payoffs. For a practical example, see "Grouping Data Used at the Same Time" on page 139.

Lazy Evaluation

Never evaluate an item until it is needed. Examples:

- In building a table of Fibonacci numbers (or any other function), populate the table with only the numbers that are actually requested, as they are requested.
- Brian Kernighan reduced the run time of [the original *troff*] by twenty percent by calculating the width of the current line when needed, rather than after each character.

Time-for-Space Rules

Packing

Dense storage representations can decrease storage costs by increasing the time to store and retrieve data. Typical examples include packing multiple binary integers into 32- or 64-bit words.

The memory available in a typical SN0 server is large enough that you might think such bit-squeezing techniques ought to be relics of the past. However, is file access time a significant factor in your program's speed? (Use *par* to find out.) If the file format can be squeezed 50%, sequential access time is cut by that much, and direct access time is reduced due to shorter seek distances. Also, if an important array does not fit into cache, and by packing a data structure you can get all of it into cache, the extra instruction cycles needed to unpack each item are repaid many times over.

Interpreters

The space required to represent a program can often be decreased by the use of interpreters in which common sequences of operations are represented compactly. A typical example is the use of a finite-state machine to encode a complex protocol or lexical format into a small table.

Finite-state machines, decision tables, and token interpreters can all yield elegant, maintainable algorithm designs. However, keep in mind that the coding techniques often used to implement an interpreter, in particular computed gotos and vector tables, tend to defeat the compiler's code manipulations. Don't expect the compiler to optimize or parallelize the interpreter's central loop.

Loop Rules

Many of these loop-transformation rules have been incorporated into the current compilers. This makes it easier to apply them in controlled ways using compiler options.

Code Motion Out of Loops

An invariant expression (one whose value does not depend on the loop variable) should be calculated once, outside the loop, rather than iteratively. But keep in mind:

1. The compiler is good at recognizing invariant expressions that are evident in the code. Place expressions where it is most natural to read or write them, and let the compiler move them for you. Example:

   ```
   for (i=0;i<Z;++i) { if (x[i]<(fact/div))...; }
   ```

   At nonzero optimization levels, the compiler will recognize `fact/div` as invariant, and move it to a generated temporary in the loop prologue. You should not modify the code to move the expression out of the loop; leave the code in this naive but readable form, and let the compiler do the work.

2. The compiler cannot recognize when a call to a user function is invariant.

   ```
   for (i=0;i<Z;++i) { if (x[i]<func(fact,div))...; }
   ```

   When `func(fact,div)` does not depend on `i`, move the call out of the loop:

   ```
   for (i=0,t=func(fact,div);i<Z;++i) { if (x[i]<t)...; }
   ```

   An exception is when you request function inlining. After the compiler inlines the function body, it may then recognize that some or all of the inlined code is invariant, and move those parts out of the loop.

Combining Tests

An efficient inner loop should contain as few tests as possible, and preferably only one. Try to combine exit conditions. Examples:

- Add a sentinel value to the last item of an unsorted vector, so as to avoid a separate test of the index variable.
- As a special case, use a sentinel byte value to signal the end of a string of bytes, avoiding the test for last-byte-index.

These optimizations can be especially helpful with the R10000 CPU, which can pipeline a single conditional branch, executing speculatively along the most likely arm. However, two conditional branches in succession can interfere with the pipeline and cause the CPU to skip cycles until the branch condition values are available. Thus if you condition an inner loop on a single test, you are likely to get faster execution.

Loop Unrolling

A large cost of some short loops is in modifying loop indexes. That cost can often be reduced by unrolling the loop.

This optimization can bring large benefits, especially in a superscalar CPU like the R10000 CPU. However, loop unrolling done manually is an error-prone operation, and the resulting code can be difficult to maintain. Fortunately, current compilers have extensive support for loop unrolling. You can control the amount of unrolling either with a compiler option or loop by loop with directives. The compiler takes care of the details and the bookkeeping involved in handling end-of-loop, while the source code remains readable.

A benefit of compiler-controlled loop unrolling is that the compiler applies optimizations like common subexpression elimination, constant propagation, code motion out of loops, and function call inlining both before and after it unrolls the loop. The optimizations work together for a synergistic effect. The software-pipelining optimization (rearranging the order of the generated machine instructions in order to maximize use of the R10000 execution units) also becomes more effective when it has a longer loop body to work on.

There are still cases in which the compiler does not recognize the opportunity to unroll loops. One is the case where the code could take advantage of the ability of the SN0 node board to maintain two or more cache miss requests simultaneously. Consider the following loop:

```
      m= 1
      DO 24  k= 2,n
         IF( X(k).LT.X(m))  m= k
   24 CONTINUE
```

This is a normal, stride-1 scan of an array. The compiler inserts prefetch instructions to overlap the fetch of one cache line with the processing of another. However, a substantial speed increase can be had by making the hardware keep two cache fetches going concurrently at all times. This is done by unrolling the loop one time, and accessing the halves of the array in two, concurrent scans, as follows:

```
      do k=2,n/2
         if(x(k).lt.x(m)) m=k
         if(x(k+n/2).lt.x(m2)) m2=k
      enddo
      if(x(m2).lt.x(m)) m=m2
```

Transfer-Driven Loop Unrolling

When a large cost of an inner loop is devoted to trivial assignments (other than maintenance of loop indices), the assignments can be reduced by duplicating the loop body and renaming the variables in the second copy.

As with simple loop unrolling, this important optimization is well-handled by the compiler. Leave your code simple and readable and let the compiler do this.

Unconditional Branch Removal

A fast loop should have no unconditional branches. An unconditional branch at the end of a loop can be handled by "rotating" the loop to put a conditional branch at the bottom.

This optimization might be needed in assembly-language code, but no modern compiler should generate code for a loop with the test at the top and an unconditional branch at the bottom.

Loop Fusion

When two nearby loops operate on the same set of elements, combine their operational parts and use only one set of loop-control operations. Example: A pair of similar loops such as the following:

```
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
      b[i][j] = a[i][j];
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
      c[i][j] = b[i][j];
```

Such a pair can be fused into a single loop such as this:

```
for (i=0;i<n;i++)
   for (j=0;j<n;j++)
      b[i][j] = a[i][j];
      c[i][j] = b[i][j];
```

This technique can produce important benefits in the SN0 architecture because, first, the contents of array *b* are only passed through the cache once, and second, the instructions to load each element of *b* (from the second loop) are eliminated. However, it is not necessary for you to perform this optimization manually. The compilers perform loop fusion automatically at certain optimization levels. The compilers can also perform the following optimizations:

- Loop interchange—exchanging the induction variables of inner and outer nested loops so as to pass over memory consecutively.
- Cache blocking—breaking a single loop nest into a sequence of smaller nests that operate on blocks of data that each fit in secondary cache.
- Loop fission—when two inner loops are controlled by a single outer loop, duplicating the outer loop and creating two simple loop nests that can then be interchanged, blocked, or unrolled.

Each of these changes, if done manually, complicates the source code with many temporary variables and insanely complicated loop control. In general, write loop code in the simplest, most portable and maintainable fashion, and then let the compiler tangle it behind the scenes.

Logic Rules

Exploit Algebraic Identities

In a conditional expression, replace a costly expression with an algebraically equivalent expression that is cheaper to evaluate. Examples:

- Not `(sqr(X)>0)` but `(X!=0)`. If this also allows you to avoid calculating `sqr(X)` unless it is needed, it is an instance of Lazy Evaluation.
- Not `(not(A) .and. not(B))` but `not(A .and. B)` (DeMorgan's Law).

When such replacements require insight, they are worth doing. Do not spend time replacing simple manifest expressions with constants; the compiler is good at that, and good at recognizing common subexpressions.

Short-Circuit Monotone Functions

When a monotone function of several variables is to be tested for a threshold value, break off the test as soon as the result is known. Short-circuit evaluation of a Boolean expression is a well-known case (handled automatically by the compiler). Bentley gave a more interesting example. Example A-1 shows a function to find the point in a list that is nearest to a given point.

Example A-1 Naive Function to Find Nearest Point

```
typedef struct point_s { double X, Y; } point_t;
#include <math.h>
int nearest_XY(int n, point_t given, point_t points[])
{
   int j, close_n=-1;
   double thisDist, closeDist=MAXFLOAT;
   for (j=0; j<n; ++j)
   {
      thisDist = sqr(points[j].X-given.X) + sqr(points[j].Y-given.Y);
      if (thisDist < closeDist)
      {
         closeDist = thisDist;
         close_n = j;
      }
   }
   return close_n;
}
```

Example A-1 as it stands short-circuits the distance computation (for purposes of comparing two distances, it is sufficient to compare the sum of squares and skip doing the square root). However, in many cases the X-distance alone is enough to eliminate a point, which avoids the need for the second multiply and an addition. Example A-2 applies this insight.

Example A-2 Nearest-Point Function with Short-Circuit Test

```
int nearest_XZ(int n, point_t given, point_t points[])
{
   int j, close_n=-1;
   double thisDist, closeDist=MAXFLOAT;
   for (j=0; j<n; ++j)
   {
      thisDist = sqr(points[j].X-given.X);
      if (thisDist < closeDist) /* possible */
      {
         thisDist += sqr(points[j].Y-given.Y);
         if (thisDist < closeDist)
         {
            closeDist = thisDist;
            close_n = j;
         }
      }
   }
   return close_n;
}
```

Reorder Tests

Logical tests should be arranged such that cheap tests precede expensive ones, and so that ones most likely to succeed precede ones more likely to fail.

In the absence of other information, the Silicon Graphics compilers assume that the `then`-leg of an `if` is more likely than the `else`, so it is good practice to put the most likely code in the then-leg. You can pass exact instruction counts in a compiler feedback file (see "Passing a Feedback File" on page 124). The compiler takes branch execution counts from the feedback file, and creates an instruction schedule that reflects the exact probabilities of each branch.

This is a convenient feature that can have good results on a poorly written program. However, the compiler does not take into account the cost of a test that involves a function call, or a test that can touch an unused cache line or page. Your insight is still needed to order tests so as to avoid expensive operations when possible.

Precompute Logical Functions

A logical function over a small, finite domain can be replaced by a table lookup. Examples include using a table to classify characters into lexical classes, and implementing an interpreter using a branch table indexed by instruction code.

When the table is small and frequently used, this technique is almost always helpful. However, when the table is sparse, or much larger than the executable code that it replaces, the effect of the table on the cache could swamp the benefits of its use.

Replace Control Variables with Control Statements

Assignment to a Boolean variable and its later test can be replaced by an if statement on the Boolean expression. Generalizing, assignment to any control variable over a small, finite domain can be replaced by a case statement on the value that would have been assigned.

Application of this rule saves three things: declaration of a variable to hold a control value; assignment into the variable; and fetching from the variable when the value is tested. The rule says that instead, you should modify the code so the control expression is used to direct execution as soon as its value is known. In some cases you have to duplicate code in order to apply the rule.

The compilers are good at reusing the stack space of local variables after their last use, and also good at saving calculated values in machine registers between their assignment and a nearby use. When the sole use of a local variable is to hold a value between its expression and a following test, the compiler (at *-O2* and higher) is quite capable of eliminating the variable and using only a register temp.

Hence the only time you should distort your program's logic to apply this rule is when the assignment is separated from its use by many statements, or by a non-inlined procedure call. (A procedure call usually makes the compiler spill register values to memory.)

Procedure Design Rules

Collapse Procedure Hierarchies

The run time of a set of procedures that (nonrecursively) call themselves can often be reduced by rewriting the procedures inline and binding what were passed arguments.

This technique is extremely useful, but difficult to apply to an existing program and yet leave the program readable and maintainable. Fortunately, current compilers contain extensive support for interprocedural analysis (IPA) and automatic inlining. Given that, it is best to write the program in a simple, well-structured style, not fretting about frequent calls to small procedures, and to let the compiler inline the small procedures.

Although inlining eliminates the instruction overhead of a function call, it inflates the size of the object code. At a certain level of inlining, the effect of the inflated, low-density code on cache and virtual memory wastes as much time as it saves.

Keep in mind the maintenance implications of inlining procedures in code that you distribute to other users. *A library procedure that has been inlined even once can never be field-replaced*. You can send out a revised version of the library, but programs that inlined the function will never call the library version.

Exploit Common Cases

A procedure should be organized to handle the most common cases with the greatest efficiency, and all remaining cases correctly (if more slowly). When designing the application, attempt to arrange the input conditions so that the efficient cases are in fact the most common.

The Caching rule (“Caching” on page 258) can be seen as an extension of this rule, in which “the commonest case” is “whatever case was most recently requested.” Bentley cites a Julian-date conversion function in which it was observed that 90% of all calls presented the same date as the previous call. Saving the last result, and returning it without recomputing when appropriate, saved time.

One way to apply the rule is to code a function so it tests for the common cases and returns their results quickly, falling through to the more involved code of the hard cases. However, a function that contains the code to handle all cases is likely to be a bulky one to inline. In order to exploit common cases and also support function inlining, write two functions. One, a public function, is short, and efficiently handles only the most common cases. Whenever it is called with a case it does not handle, it calls a longer, private function to deal with all remaining cases. The shorter function can be inlined (at multiple places, if necessary) without greatly inflating the code. The longer should not be inlined.

Use Coroutines

A multiphase algorithm in which the phases are linked by temporary files (or arrays) can be reduced to a single-pass algorithm using coroutines.

Coroutines are defined and described in detail in Knuth (Volume I) and most other modern books on algorithmics. Under IRIX, you can write coroutines in a natural way using any one of three models:

- The UNIX model of forked processes that communicate using pipes.
- The POSIX threads model using POSIX message queues to communicate.
- The MPI (or PVM) model of cooperating processes exchanging messages.

Transform Recursive Procedures

A simple, readable solution expressed as a recursion can be transformed into a more efficient iterative solution. You can:

- Code an explicit recursion using a stack variable.
- When the final action of a procedure is to call itself ("tail recursion"), replace the call with a **goto** to the top of the procedure. (Then rewrite the iteration as a loop.)
- Cut off the recursion early for small arguments (for example using a table lookup), instead of recurring down to the case of size 0.

The compilers recognize simple cases of tail recursion and compile a **goto** to the top of the function, avoiding stack use. More complicated recursion defeats most of the compiler's ability to modify the source: recursive procedures are not inlined; a loop containing a recursion is not unrolled; and a loop with a recursion is not executed in parallel. Hence there is good reason to remove recursion, after it has served its purpose as a design tool.

Use Parallelism

Structure the program to exploit the parallelism available in the underlying hardware.

This rule is certainly a no-brainer for the SN0 systems; parallel execution is what they do best. However, you can apply parallelism using a variety of programming models and at a wide range of scales from small to large. Two examples among many:

- Using the POSIX compliant asynchronous I/O library, you can schedule one or an entire list of file accesses to be performed in a parallel thread, with all the file-synchronous waits being taken by a different process.
- The native integer in the MIPS IV ISA has 64 bits (under either -n32 or -64 compiler option). You can pack many smaller units into `long long` (in Fortran, `INT*8`) variables and operate on them in groups.

Parallel algorithm design is a very active area in computer science research. Execution of the query `title:parallel & algorithm* & library` on the Alta Vista search engine (as a Boolean expression, in the Advanced Search) produced numerous hits.

Expression Rules

Initialize Data Before Runtime

As many variables as possible (including arrays and tables) should be initialized befor program execution.

This rule encompasses the use of compile-time initialization of variables—in C, by use of initializing expressions in declarations, in Fortran by use of the PARAMETER clause—and also the preparation of large arrays and tables.

Compile-time initialization with manifest constants allows the compiler greater scope. The compiler precalculates subexpressions that have only constant arguments. When IPA is used, the compiler recognizes when a procedure argument is always a given constant, and propagates the constant into the procedure body. When functions are inlined, constants are propagated into the inlined code, and this produces still more constant expressions for the compiler to eliminate.

Some programs spend significant time initializing large arrays, tables, and other data structures. There are two common cases: clearing to zero, and processing initial data from one or more files.

Initializing to Zero

A for-loop that does nothing but store zero into an array is a waste of CPU cycles. Worse, in IRIX on SN0 systems, all memory is by default allocated to the node from which it is first touched, which means that the zero array will be allocated in the node where the for-loop runs. So at the least, make sure that each parallel thread of the program zeros the array elements that it uses, and no others, so that memory is allocated where it is used.

In a C program, it is much more efficient to use **calloc** (see the calloc(3) reference page) to allocate memory filled with zero. The page frames are defined, but the actual pages are not created and not filled at this time. Instead, zero-filled pages are created as the program faults them in. This ensures that no time is spent clearing pages that are never used, and that pages are allocated to the node that actually uses them.

A similar alternative is to use **mmap** (see the mmap(2) reference page) to map an extent of the pseudo-device */dev/zero*. The mapped extent is added to the program's address space, but again, zero-filled pages are created and allocated as they are faulted.

In a Fortran program, it is somewhat more efficient to call the C library function **bzero()** than to code a DO loop storing zero throughout an array. However, this does touch the cleared pages.

Initializing from Files

The time to read and process file data is essentially wasted because it does not contribute to the solution. Bentley recommended writing a separate program that reads the input data and processes it into some intermediate form that can be read and stored more quickly. For example, if the initializing data is in ASCII form, a side program can preprocess it to binary. When the side program uses the same data structures and array dimensions as the main program, the main program can read the processed file directly into the array space.

IRIX permits you to extend this idea. You can use **mmap** (see the mmap(2) reference page) to map any file into memory. You can take all the code that reads files and initializes memory arrays and structures, and isolate it in a separate program. This program maps a large block of memory to a new disk file. It reads the input file data, processes it in any way required, and builds a complete set of data structures and arrays of binary data in the mapped memory. Then the program unmaps the initialized memory (or simply terminates). This writes an image of the prepared data into the file.

When the main program starts up, it maps the initialized file into memory, and its initialization is instantly complete. Pages of memory are faulted in from the disk file as the program refers to them, complete with the binary content as it was prepared by the initializing program. This approach is especially handy for programs that need large in-memory databases, like the scenery files used in a visual simulator.

Exploit Algebraic Identities

Replace the evaluation of a costly expression by one that is algebraically equivalent but less costly. For example: not `ln(A)+ln(B)` but `ln(A*B)` (consider that `A*B` might overflow).

Bentley mentioned that the frequent need to test whether an integer J is contained in the inclusive range (I,K), naively coded `((I<=J)&&(J<=K))`, can be implemented as `((J-I)<(K-I))`. The value (K-I) can be precomputed. (When I and K are constants or are propagated from constant expressions, the compiler precomputes it automatically.)

you compile with the *IEEE_arithmetic=3* and *roundoff=3* options, the compilers automatically perform a number of mathematically valid transformations to make arithmetic faster.

Algebraic identities are at the heart of strength reduction in loops. The compiler can recognize most opportunities for strength reduction in an expression that includes the loop induction variable as an operand. However, strength reduction is a general principle, and the compiler cannot recognize every possibility. Examine every expression in a loop to see if the same value could not be calculated more cheaply by an incremental change in the same value from the previous iteration of the loop.

Eliminate Common Subexpressions

When the same expression is evaluated twice with no assignments to its variable operands, save the first evaluation and reuse it.

This rule, application of which is almost second nature to experienced programmers, is in fact applied well by the compilers. The compilers look for common subexpressions both before and after constant propagation and inlining. Often they find subexpressions that are not superficially apparent in the source code. The expression value is held for its second use in a machine register if possible, or the compiler may create a temporary stack variable to hold it.

Combine Paired Computation

When similar expressions are evaluated together, consider making a new function that combines the evaluation of both.

Bentley cites Knuth as reporting that both sine and cosine of an angle can be computed as a pair for about 1.5 times the cost of computing one of them alone. The maximum and minimum values in a vector can be found in one pass for about 1.5 times the cost of finding either. (The current compilers can perform this particular optimization automatically. See the opt(5) reference page, the *cis*= parameter.)

When it is not practical for the function to return both values, it can at least cache the paired answer (see "Caching" on page 258) to be returned on request. In the case of sine and cosine, each of the functions can test to see if its argument is the same as the previous angle passed to either function. When it is not, call the function that calculates and caches both sine and cosine. Return the requested result.

In this regard, note that the compilers do recognize the dual nature of integer division and remainder, so that the sequence `{div=a/b;rem=a%b;}` compiles to a single integer divide.

Exploit Word Parallelism

Use the full word width of the underlying computer architecture to evaluate binary values in parallel.

This is a special case exploiting parallelism. The native integer in all current MIPS CPUs is 64 bits. The standard Fortran Boolean functions (IOR, IAND, and so on) handle INTEGER*8 (that is, 64-bit) values.

Appendix B

R10000 Counter Event Types

The MIPS R10000 CPU contains two 32-bit hardware counters, each of which can be assigned to count any one of 16 events. The counters can be used in two ways. First, they can be used to tabulate the frequency of events in a particular program, for example, counting all instructions, or counting floating-point instructions.

Second, the CPU can be conditioned to take a trap when one of the counters overflows. If the counter is preloaded to be just *N* short of an overflow, the CPU will trap when exactly *N* events have occurred. The *ssrun* command uses this feature to sample a program's state at regular intervals; for example, every 32K graduated instructions (see "Sampling through Hardware Event Counters" on page 65).

The IRIX kernel extends the utility of the counters by giving each process its own set of virtual counter registers. When the kernel preempts a process, it saves its current counter registers, just as it saves other machine registers. When the kernel resumes execution of a process, it restores the counter values along with the rest of the machine registers. In this way, every process can accumulate its own counts accurately, even though it shares the CPU with the kernel and many processes. (See reference page r10k_counters(5).)

Table B-1 summarizes the types of events that can be counted by the R10000 CPU in either Counter 0 or Counter 1. The table is ordered by the event type number as used in the R10000 special register that controls event counting. The same event numbers are used by the *perfex* and *ssrun* commands (see "Profiling Tools" on page 53). A detailed discussion of the events follows the table.

Table B-1 R10000 Countable Events

Event Number	Counter 0 Event	Event Number	Counter 1 Event
0	Cycles	16	Cycles
1	Instructions issued to functional units	17	Instructions graduated
2	Memory data access (load, prefetch, sync, cacheop) issued	18	Memory data loads graduated
3	Memory stores issued	19	Memory data stores graduated
4	Store-conditionals issued	20	Store-conditionals graduated
5	Store-conditionals failed	21	Floating-point instructions graduated
6	Branches decoded	22	Quadwords written back from L1 cache
7	Quadwords written back from L2 cache	23	TLB refill exceptions
8	Correctable ECC errors on L2 cache	24	Branches mispredicted
9	L1 cache misses (instruction)	25	L1 cache misses (data)
10	L2 cache misses (instruction)	26	L2 cache misses (data)
11	L2 cache way mispredicted (instruction)	27	L2 cache way mispredicted (data)
12	External intervention requests	28	External intervention request hits in L2 cache
13	External invalidate requests	29	External invalidate request hits in L2 cache
14	Instructions done (in chip rev 2.x, virtual coherence)	30	Stores, or prefetches with store hint, to CleanExclusive L2 cache blocks
15	Instructions graduated	31	Stores, or prefetches with store hint, to Shared L2 cache blocks

Counter Events In Detail

This section lists the events in related groups for clarity. There are a few bugs in the counting algorithms in early runs of the R10000 chip (through revisions 2.x), so counts can differ between systems even when all other factors are the same. Roughly speaking, R10000 revision 2.x chips are used in machines manufactured before 1997, and revision 3.2 in 1997 and later.

Clock Cycles

Either counter can be incremented on each CPU clock cycle (event 0 or event 17). This permits counting cycles along with any other event. Note that a 32-bit counter overflows after only 21 seconds at 200 MHZ.

Instructions Issued and Done

Several counters record when instructions of different kinds are "issued," that is, taken from the input queue and assigned to a functional unit for processing. Some instructions can be issued more than once before they are complete, and some can be issued and then discarded (speculative execution). As a result, issued instructions reflect the amount of the work the CPU does, but only graduated instructions (see "Graduated Instructions" following) reflect the effective work toward completion of the algorithm.

Issued Instructions (Event 1)

This counter is incremented on each cycle by the sum of these factors:

- Integer operations marked as "done" in the active list. Zero, 1, or 2 operations can be so marked on each cycle.
- Floating point operations issued to an FPU. Zero, 1, or 2 can be issued per cycle. In early revs of the chip, a conditionally or tentatively issued FP instruction can be counted as issued multiple times.
- Load and store instructions issued to the address calculation unit on the previous cycle: 0 or 1 per cycle. In early chips, prefetch instructions are not counted as issued, and loads and stores are counted each time they are issued to the address calc unit, which can be multiple times per instruction.

A load/store instruction can be reissued if it does not complete its tag check cycle. One case occurs when the data cache tag array is busy for the required bank because of an external operation (a refill or an invalidate cycle), or if the CPU initiated a refill on the previous cycle to the same bank. This case usually generates just one extra "issue."

Another case occurs when the Miss Handling Table is already busy with four operations and cannot accept another. In this case, the instruction is re-issued up to once every four cycles as long as the Miss Handling Table is full. If several instructions are waiting, a spurious issue could be counted every cycle.

Issued Loads (Event 2)

This counter is incremented when a load instruction was issued to the address-calc unit on the previous cycle. Unlike the combined count in Event 1, this counts each load instruction only once. Prefetches are counted with issued loads since revision 3.x. See the discussion of "Issued Versus Graduate Loads and Stores" on page 277.

Issued Stores (Event 3)

This counter is incremented when a store instruction was issued to the address-calc unit on the previous cycle. Store-conditional instructions are included in this count. Unlike the combined count in Event 1, this counts stores as issued only once. See also "Issued Store Conditionals (Event 4)" on page 284, and the discussion of "Issued Versus Graduate Loads and Stores" on page 277.

Instructions Done (Event 14)

Beginning with chip revision 3.x, this counter's meaning is changed. It is incremented on the cycle after either ALU1, ALU2, FPU1, or FPU2 marks an instruction as "done." Done is not the same as graduated, because an instruction that is done, while complete, can still be discarded if it is on a speculative branch line that was mispredicted.

This counter had a different meaning in early revisions of the chip, see "Virtual Coherency Conditions (Old Event 14)" on page 283.

Graduated Instructions

An instruction "graduates" when it is complete and can no longer be discarded. The R10000 CPU *issues* instructions whenever it can, and it *executes* instructions in any sequence it can (and sometimes executes instructions on speculation only to discard them). However, it *graduates* instructions in the sequence they were written, so that graduation means that an instruction has had its final effect on the visible state of registers and memory. This predictability makes graduated instructions a more reliable, repeatable count of the instructions executed by a program than the issued-instructions counts. An ideal profile run, which counts executed instructions by software, should agree with the count of graduated instructions for the same program and input. (However, revision 2.x chips can slightly undercount graduated instructions, as compared to an ideal profile run.)

Issued Versus Graduate Loads and Stores

Load and store instructions (and prefetch, load conditional, and store conditional) all require address translation and possibly cache operations. These instructions can be issued, delayed, retracted, and reissued before they are finished and graduate. For this reason, issued loads and stores always outnumber graduated loads and stores. However, the difference between a load or store issued and one graduated can provide valuable insight into performance problems.

Specifically, after a load or store is issued it can often be killed due to contention for some on-chip or off-chip resource, such as a tag-bank read-port. In that case the instruction is reissued, and so counted twice by the issued counter but only once by the graduated counter.

Normally reissues are rare, so issued and graduated should correspond reasonably well. But sometimes resource contention can become a performance bottleneck, and in that case the number of issued load/stores can soar. If you see the number of issued loads or stores greatly exceeding the number graduated, look at the count of mispredicted branches. If it remains low, you can guess that the load/store pipeline is having some kind of resource contention, causing those instructions to be issued repeated. The R10000 CPU can issue at most one load or store per cycle, so excessive reissues can seriously degrade performance.

Graduated Instructions (Event 15)

This counter is incremented by the number of instructions that were graduated on the previous cycle. Integer multiply and divide instructions are counted as two instructions.

Graduated Instructions (Event 17)

Same as Event 15. Supporting this count in either counter register permits counting graduated instructions along with any other value.

Graduated Loads (Event 18)

This counter is incremented by the number of loads that graduated on the previous cycle. Prefetch instructions are included in this count. Up to four loads can graduate in one cycle. (Through revision 2.x, when a store graduates on a given cycle, loads that graduate on that cycle do not increment this counter.)

Graduated Stores (Event 19)

This counter is incremented on the cycle after a store graduates. At most one store can graduate per cycle. Prefetch-exclusive instructions (which indicate an intent to modify memory) are counted here, as are store conditional instructions (see "20 Graduated store conditionals" on page 285).

Graduated Floating Point Instructions (Event 21)

This counter is incremented by the number of floating point instructions that graduated on the previous cycle. There can be 0 to 4 such instructions. A multiply-add counts as just one floating-point instruction.

Branching Instructions

The R10000 CPU predicts whether any branch will be taken before the branch is issued. Execution continues along the predicted line. The instructions executed speculatively can be done—incrementing event 14, see "Instructions Done (Event 14)" on page 276—but if the prediction proves to be false, they are discarded and do not increment event 15 ("Graduated Instructions (Event 15)" on page 278)

Decoded Branches (Event 6)

In old chips (revision 2.x), this counter is incremented when any branch instruction is decoded. This includes both conditional and unconditional branches. Although conditional branches have been predicted at this point, they may still be discarded due to an exception or a prior mispredicted branch.

Starting with revision 3.x, this counter is incremented when a conditional branch is determined to have been correctly or incorrectly predicted. This occurs once for every branch that the program actually executes. The count in current chips does not include unconditional branches, and it does not include branches that are decoded on speculation and then discarded. The current definition of the counter enables meaningful comparison with event 24 ("Mispredicted Branches (Event 24)" on page 279).

The determination of prediction correctness is known as the branch being "resolved." Some branches depend on conditions in the floating-point unit. Multiple floating-point branches can be resolved in a single cycle, but this counter can be incremented by only 1 in any cycle. As a result, when multiple FP conditional branches are resolved in a single cycle, this count will be incorrect (low). This is a relatively rare event.

Mispredicted Branches (Event 24)

This counter is incremented on the cycle after a branch is restored because it was mispredicted.

The R10000 CPU should predict a high percentage of branches correctly. The compilers order branches based on heuristic evaluation of their likelihood of being taken, or on the basis of a feedback file created from a trace. If the count of event 24 is more than a few percent of event 6 ("Decoded Branches (Event 6)" on page 279), something is wrong with the compiler options or something is unusual about the algorithm. You can use *prof* to generate a feedback file containing actual branch frequency, and the compiler can use such a feedback file to order the machine instructions to minimize mispredictions. (See "Passing a Feedback File" on page 124.)

Primary Cache Use

The R10000 primary cache consists of separate instruction cache (i-cache) and data cache (d-cache) contained on the CPU chip. The primary caches together are called the level-one (L1) cache. This small area (2×32KB) is organized as 16-byte units called quadwords. Instructions and data are fetched and stored between the primary cache and the secondary cache in quadwords. Several counters document L1 cache use.

Primary Instruction Cache Misses (Event 9)

This counter is incremented on the cycle after a request to refill a line of the primary instruction cache is entered into the Miss Handling Table. The count indicates that a needed instruction word was not in the primary cache. The relationship between this counter and event 1 (“Issued Instructions (Event 1)” on page 275) indicates the effectiveness of the L1 i-cache.

Primary Data Cache Misses (Event 25)

This counter is incremented one cycle after a request to refill a line of the primary data cache is entered into the Miss Handling Table. The count indicates that a needed operand was not in the primary cache. The affected instruction is suspended and reissued when the needed quadword arrives.

Quadwords Written Back from Primary Data Cache (Event 22)

This counter is incremented on each cycle that a quadword of data is valid and being written from primary data-cache to secondary cache. Quadwords are written back only when they have been modified by a store.

Secondary Cache Use

The secondary cache, also called the level-two (L2) cache, is external to the R10000 chip. MIPS documentation for the CPU does not describe the L2 cache in detail because its design is not dictated by the CPU chip. The design of the L2 cache is part of the total system design. For example, the L2 cache of the POWER CHALLENGE 10000 system is very different from the L2 cache of the SN0 systems. The counters for L2 cache events are defined in CPU terms, not in terms of the cache design.

Quadwords Written Back from Scache (Event 7)

This counter is incremented on each cycle that the data for a quadword is written back from the secondary cache to the system-interface unit. In SGI systems, the L2 cache is organized as 128-byte lines. One cache line is 8 quadwords, so this counter is updated by 8 on each cache line writeback.

Correctable Scache Data Array ECC Errors (Event 8)

The interface between the CPU chip and the L2 cache includes lines for an ECC code. This counter is incremented on the cycle following the correction of a single-bit error in a quadword read from the secondary cache data array. A small number of single-bit errors is inevitable in high-density logic systems. However, any significant count in this counter is cause for concern.

Secondary Instruction Cache Misses (Event 10)

This counter is incremented the cycle after the fourth quadword of a cache line is written from memory into the secondary cache, when the cache refill was initially triggered by a primary instruction cache miss. Data misses are counted in "Secondary Data Cache Misses (Event 26)."

Instruction Misprediction from Scache Way Prediction Table (Event 11)

This counter is incremented when the secondary cache control begins to retry an access because it hit in the "way," or bank, that was not predicted, and the event that initiated the access was an instruction fetch. Data mispredictions are counted in "Data Misprediction from Scache Way Prediction Table (Event 27).".

Secondary Data Cache Misses (Event 26)

This counter is incremented the cycle after the second quadword of a cache line is written from memory into the secondary cache, when the cache refill was initially triggered by a primary data cache miss. Instruction misses are counted in "Secondary Instruction Cache Misses (Event 10)."

Data Misprediction from Scache Way Prediction Table (Event 27)

This counter is incremented when the secondary cache control begins to retry an access because it hit in the "way," or bank, that was not predicted, and the event that initiated the access was not an instruction fetch. Instruction mispredictions are counted in "Instruction Misprediction from Scache Way Prediction Table (Event 11)."

Store or Prefetch-Exclusive to Clean Block in Scache (Event 30)

This counter is incremented on the cycle after an update request is issued for a clean line in the secondary cache. An update request is the result of processing a store instruction or a prefetch instruction with the exclusive option. In the SN0, the update request from this CPU will cause the hub chip to update memory and possibly to initiate other cache coherency operations (see "Cache Coherency Events" on page 282).

Store or Prefetch-Exclusive to Shared Block in Scache (Event 31)

This counter is incremented on the cycle after an update request is issued for a shared line in the secondary cache. An update request is the result of processing a store instruction or a prefetch instruction with the exclusive option. In the SN0, the update request from this CPU will cause the hub chip to update memory and possibly to initiate other cache coherency operations (see "Cache Coherency Events" on page 282). For example, if other CPUs have a copy of the modified cache line, they will be sent invalidations ("External Invalidations (Event 13)" on page 283).

This event is the best indication of cache contention between CPUs. The CPU that accumulates a high count of event 31 is repeatedly modifying shared data.

Cache Coherency Events

In a cache-coherent multiprocessor, the system signals to the CPU when the CPU has to take action to maintain the coherency of the primary and secondary cache data. In the SN0 architecture, it is the hub chip that signals the CPU when some other CPU has invalidated memory (see "SN0 Node Board" on page 10 and "Understanding Directory-Based Coherency" on page 13).

From the standpoint of the CPU, an "intervention" is a signal stating that some other CPU in the system wants to use data from a cache line that this CPU has. (Other CPUs can tell from the directory bits stored in memory that this CPU has a copy of a cache line; see "Understanding Directory-Based Coherency" on page 13). The other CPU requests the status of the cache line, and requests a copy of the line if it is not the same as memory.

An "invalidation" is a notice that another CPU has modified memory data that this CPU has in its cache. This CPU needs to invalidate (that is, discard) its copy of the data.

External Interventions (Event 12)

This counter is incremented on the cycle after an intervention is received and the intervention is not an invalidate type.

External Invalidations (Event 13)

This counter is incremented on the cycle after an external invalidate is entered.

Virtual Coherency Conditions (Old Event 14)

This is an obsolete counter definition valid only through chip revision 2.x. Beginning with revision 3.x, counter 14 has a different meaning (see “Instructions Done (Event 14)” on page 276).

External Intervention Hits in Scache (Event 28)

This counter is incremented on the cycle after an external intervention is determined to have hit in the L2 cache, necessitating a response of status and possibly a copy of the cache line.

External Invalidation hits in Scache (Event 29)

This counter is incremented on the cycle after an external invalidate request is determined to have hit in the L2 cache, necessitating an action.

This event is a good indicator of cache contention. The CPU that produces a high count of event 29 is being slowed because it is using shared data that is being updated by a different CPU. The CPU doing the updating generates event 31 (“Store or Prefetch-Exclusive to Shared Block in Scache (Event 31)” on page 282).

Virtual Memory Use

The CPU uses a table, the translation lookaside buffer (TLB), to map virtual addresses to physical addresses. The TLB points to a limited number of pages. When a virtual address cannot be found in the TLB, the CPU traps to a fixed location. Operating system code in real memory analyzes the miss against the page tables in memory that describe the process's full address space. If the virtual address is valid, the trap code eventually loads one or more new entries into the TLB and resumes execution.

A TLB miss involves at least some in-memory processing. It can precipitate extensive processing to write pages out, allocate a page frame, and read a page from backing store. For this reason, the number of TLB misses, averaged per second of the program's run (after initial startup transients), is an important performance metric.

TLB Misses (Event 23)

This counter is incremented on the cycle after the TLB miss handler is invoked.

Lock-Handling Instructions

The Load Linked instruction and Store Conditional instruction (LL/SC) are used to implement mutual exclusion objects such as locks and semaphores. A Load Linked instruction loads a word from memory and simultaneously tags its cache line in memory. The matching Store Conditional tests the target cache line: if it has not been updated since the Load Linked was executed, the Store Conditional succeeds and modifies memory, removing the tag from the cache line. If the target line has been modified, the Store Conditional fails. The pair of instructions can be used to implement various kinds of mutual exclusion.

LL/SC should never be significant in program execution time—when they are, it indicates some kind of contention or false-sharing problem involving mutual exclusion between asynchronous threads.

Issued Store Conditionals (Event 4)

This counter is incremented when a store-conditional instruction was issued to the address-calc unit on the previous cycle. By subtracting this count from event 3 ("Issued Stores (Event 3)" on page 276), you can isolate nonconditional (normal) store instructions. This counter cannot count a given instruction more than once.

Failed Store Conditionals (Event 5)

This counter is incremented when a store-conditional instruction fails; that is, when the target cache line had been modified between the completion of the Load Linked and the execution of the Store Conditional. A failed instruction also graduates and is counted in event 20 (“20 Graduated store conditionals” on page 285).

A small proportion of failed SC instructions is to be expected when asynchronous threads use mutual exclusion. However, anything more than a few percent of failures indicates a performance problem, either because the shared resource is overused (bad design) or because the target cache line is occupied by too many modifiable variables (false sharing).

20 Graduated store conditionals

This counter is incremented on the cycle following the graduation of any store-conditional instruction, including one counted in event 5 (“Failed Store Conditionals (Event 5)” on page 285).

Appendix C

Useful Scripts and Code

This appendix contains the following example programs, shell scripts, and awk scripts that are used to create some of the examples in this book:

- "Program adi2" on page 288 is an example Fortran program used to demonstrate problems in cache and TLB use.
- "Basic Makefile" on page 292 is the skeleton of a makefile that handles compiler options in different categories.
- "Software Pipeline Script swplist" on page 294 is a shell script that compiles a module to create an assembly listing and extracts the software pipeline report cards.
- "Shell Script ssruno" on page 297 is a simple script to make SpeedShop experiments more convenient to run.
- "Awk Script for Perfex Output" on page 298 demonstrates how the output of *perfex* can be post-processed for analysis.
- "Awk Script for Amdahl's Law Estimation" on page 300 reads execution times, derives the parallel fraction of the program, and extrapolates the execution time for larger numbers of CPUs based on Amdahl's law.
- "Page Address Routine va2pa()" on page 302 is a C function to return the physical address of a virtual memory variable.

Program adi2

The program *adi2* in Example C-1 is used as an example in several chapters.

Example C-1 Program adi2.f

```
program fake_adi
      implicit none
      integer ldx, ldy, ldz, nx, ny, nz, maxsteps
      parameter (ldx = 128, ldy = 128, ldz = 128)
      parameter (nx  = 128, ny  = 128, nz  = 128)
      parameter (maxsteps = 2)
      real*8 data(ldx,ldy,ldz)
      integer i, j, k, istep
      external rand, dtime
      real*4 dtime, t, t2(2)
      real*8 rand, checksum
c
      do k = 1, nz
         do j = 1, ny
            do i = 1, nx
               data(i,j,k) = rand()
            enddo
         enddo
      enddo
c
      t = dtime(t2)
c
      do istep = 1, maxsteps
c
c*$* assert concurrent call
         do k = 1, nz
            do j = 1, ny
               call xsweep(data(1,j,k),1,nx)
            enddo
         enddo
c
c*$* assert concurrent call
         do k = 1, nz
            do i = 1, nx
               call ysweep(data(i,1,k),ldx,ny)
            enddo
         enddo
```

```
c
c*$* assert concurrent call
         do j = 1, ny
            do i = 1, nx
               call zsweep(data(i,j,1),ldx*ldy,nz)
            enddo
         enddo
c
      enddo
c
      t = dtime(t2)
      write(6,1) t
 1    format(1x,'Time:     ',f6.3,' seconds')
      checksum = 0.0d0
      do k = 1, nz
         do j = 1, ny
            do i = 1, nx
               checksum = checksum + data(i,j,k)
            enddo
         enddo
      enddo
c
      write(6,2) checksum
 2    format(1x,'Checksum: ',1pe17.10)
c
      end
c---------------------------------------------------------------------
      subroutine xsweep(v,is,n)
      implicit none
      real*8 v(1+is*(n-1))
      integer is, n
      integer i
      real*8 half
      parameter (half = 0.5d0)
c
      do i = 2, n
         v(1+is*(i-1)) = v(1+is*(i-1)) + half*v(1+is*(i-2))
      enddo
c
      do i = n-1, 1, -1
         v(1+is*(i-1)) = v(1+is*(i-1)) - half*v(1+is*i)
      enddo
c
      return
      end
```

```
c--------------------------------------------------------------------
      subroutine ysweep(v,is,n)
      implicit none
      real*8 v(1+is*(n-1))
      integer is, n
      integer i
      real*8 half
      parameter (half = 0.5d0)
c
      do i = 2, n
         v(1+is*(i-1)) = v(1+is*(i-1)) + half*v(1+is*(i-2))
      enddo
c
      do i = n-1, 1, -1
         v(1+is*(i-1)) = v(1+is*(i-1)) - half*v(1+is*i)
      enddo
c
      return
      end
c--------------------------------------------------------------------
      subroutine zsweep(v,is,n)
      implicit none
      real*8 v(1+is*(n-1))
      integer is, n
      integer i
      real*8 half
      parameter (half = 0.5d0)
c
      do i = 2, n
         v(1+is*(i-1)) = v(1+is*(i-1)) + half*v(1+is*(i-2))
      enddo
c
      do i = n-1, 1, -1
         v(1+is*(i-1)) = v(1+is*(i-1)) - half*v(1+is*i)
      enddo
c
      return
      end
```

Program *adi5.f* is identical to Example C-1 except for the line shown in bold type in Example C-2.

Example C-2 Program adi5.f

```
program fake_adi
      implicit none
      integer ldx, ldy, ldz, nx, ny, nz, maxsteps
      parameter (ldx = 129, ldy = 129, ldz = 128)
      parameter (nx  = 128, ny  = 128, nz  = 128)
      parameter (maxsteps = 2)
      real*8 data(ldx,ldy,ldz)
```

Program adi53.f is identical to Example C-1 except for the lines shown in bold in Example C-3.

Example C-3 Program adi53.f

```
program fake_adi
c
      implicit none
c
      integer ldx, ldy, ldz, nx, ny, nz, maxsteps
      parameter (ldx = 129, ldy = 129, ldz = 128)
      parameter (nx  = 128, ny  = 128, nz  = 128)
      parameter (maxsteps = 2)
...
      do j = 1, ny
            call copy(data(1,j,1),ldx*ldy,temp,nx,nx,nz)
            do i = 1, nx
               call zsweep(temp(i,1),nx,nz)
            enddo
            call copy(temp,nx,data(1,j,1),ldx*ldy,nx,nz)
         enddo
...
subroutine copy(from,lf,to,lt,nr,nc)
      implicit none
      real*8 from(lf,nc), to(lt,nc)
      integer lf, lt, nr, nc
      integer i, j
      do j = 1, nc
         do i = 1, nr
            to(i,j) = from(i,j)
         enddo
      enddo
      return
      end
```

Basic Makefile

This Makefile is a template for a Makefile suitable for any moderately complex program composed of Fortran and C source files. It isolates compiler options into groups for easy editing and experimentation.

Example C-4 Basic Makefile

```
#! /usr/sbin/smake
# ------------------------------------------------------------------
# Basic Makefile for a program composed of Fortran and C modules
# ------------------------------------------------------------------
# The following variables specify the compiler and linker options,
# assembling them by groups for use in later commands. You may
# need to edit these lines several times while tuning.
#
#  -- flags related to ISA, ABI, and model (ipxx) go to $ARCH
#     -- set -n32 or -64. -TARG, -TENV could go here too.
ABI     = -n32
#     -- probably -mips4
ISA     = -mips4 -r10000
#     -- ip27 for Origin2000/Onyx2
PROC    = ip27
ARCH    = $(ABI) $(ISA)
#  -- flags related to optimization level go to $OPT
#     -- set level, e.g. -O0 g3, -O3, -Ofast=$(PROC)
OLEVEL  = -O2
#     -- set -OPT: option group
OOPT    = -OPT:alias=restrict
#     -- set -IPA: option group
OIPA    =
#     -- set -LNO: option group
OLNO    =
OPT     = $(OLEVEL) $(OOPT) $(OIPA) $(OLNO)
# -- flags related to numeric precision, by compiler
FOPTS   = -OPT:IEEE_arithmetic=3:roundoff=2
COPTS   = -OPT:IEEE_arithmetic=3:roundoff=2
# Assemble the f77 and cc flags into single variables
FFLAGS  = $(ARCH) $(OPT) $(FOPTS)
CFLAGS  = $(ARCH) $(OPT) $(COPTS)
# Link-time flags must include ABI, ISA, and opt flags
LDFLAGS = $(ARCH) $(OPT)
# ------------------------------------------------------------------
```

```
# The following variables specify the program components.
# You typically edit these lines only once, to specify the modules.
#
#   -- Specify the name of the executable program:
EXEC    = execname
#   -- list all Fortran object files, e.g. FOBJS = f1.o f2.o
FOBJS   =
#   -- list all C object files, e.g. COBJS = c1.o c2.o c3.o
COBJS   =
#   -- List all linked libs
LIBS    = -lfastm -lm
# The program comprises the following object files:
OBJS    = $(FOBJS) $(COBJS)
# ---------------------------------------------------------------------
# The following variables locate tools based on an environment
# variable (or command-line argument) $TOOLROOT.
FC      = $(TOOLROOT)/usr/bin/f77
CC      = $(TOOLROOT)/usr/bin/cc
LD      = $(FC)
F77     = $(FC)
# Locate a script that processes the .S output files
SWP     = swplist
# Shorthand for "rm" for use in "make clean"
RM      = /bin/rm -f
# ---------------------------------------------------------------------
# Nothing below this point should need editing.
# ---------------------------------------------------------------------
# The following target implements "make clean"
clean:
        $(RM) $(EXEC) $(OBJS)
# ---------------------------------------------------------------------
# The following target implements "make execname" by linking all
# all object files:
$(EXEC):        $(OBJS)
        $(LD) -o $@ $(LDFLAGS) $(OBJS) $(LIBS)
# ---------------------------------------------------------------------
# The following targets tell how to compile objects from sources.
# Variable $DEFINES is set on the make command line, if at all.
.SUFFIXES: .o .F .c .f .swp
.F.o:
        $(FC)  -c $(FFLAGS) $(DEFINES) $<
.f.o:
        $(FC)  -c $(FFLAGS) $(DEFINES) $<
.c.o:
        $(CC)  -c $(CFLAGS) $(DEFINES) $<
```

```
# ----------------------------------------------------------------------
# The following targets implement "make sourcename.swp" to inspect
# the SWP code generation (requires swplist script)
.F.swp:
        $(SWP) -c $(FFLAGS) $(DEFINES) -WK,-cmp=$*.m $<
.f.swp:
        $(SWP) -c $(FFLAGS) $(DEFINES) -WK,-cmp=$*.m $<
.c.swp:
        $(SWP) -c $(CFLAGS) $(DEFINES) $<
```

Software Pipeline Script swplist

This complex *csh* script compiles one or more C or Fortran source files with the -S option, which produces only an assembler listing, not an object file. Then it processes each of the listing files, extracting just the software pipeline "report cards," and merges these back into the original source files. The merged files, showing pipeline statistics above the loops to which they apply, are written with *.swp* extensions.

Note that the source line number the compiler assigns to a generated loop is only approximate because the higher levels of optimization transform the code. As a result, a report card in the *.swp* file sometimes precedes the loop to which it applies, although the report card sections appear in the correct sequence.

Example C-5 Shell Script swplist

```
#!/bin/csh -f
if ( $#argv == 0 ) then
    echo ""
    echo "Usage: $0 [compiler flags] files..."
    echo "    This version of the script uses the Environment variable"
    echo "    TOOLROOT if set."
    echo "    All tools are called as "\$"TOOLROOT/usr/bin/<tool>."
    exit
endif
set t = /usr/tmp
if (${?TMPDIR}) then
   if (-e ${TMPDIR}) then
      set t = ${TMPDIR}
   endif
endif
if ( ! $?TOOLROOT ) then
    setenv TOOLROOT /
endif
echo 'TOOLROOT is "'$TOOLROOT'"'
```

```
set nawk_file1 = $t/$$.SWP.NAWK_1
set nawk_file2 = $t/$$.SWP.NAWK_2
# First awk program extracts SWP descriptive lines and saves
# in temp files, one per loop. Output is a list of loop-files.
cat << NAWK_FILE1_END > $nawk_file1
BEGIN {
        Loop = 0;
        GotLine = 0;
        LoopID = 0;
        TmpFileRoot = sprintf("$t/%s_SWP",FILENAME)
}
/#<swps>/ || /#<swpf>/ {
        if (Loop == 0) {
            Loop = 1;
            LoopID++;
            TmpFile = TmpFileRoot"."LoopID;
        }
        print > TmpFile;
}
/oop line/ {
        if (Loop == 1) {
                if (GotLine == 0) {
                        GotLine = 1;
                        split(\$0, Line);
                        i=0;
                        while (Line[i] != "line") {i++}
                        LoopLine = Line[++i];
                        print LoopLine " " TmpFile
                }
        }
}
!/#<swps>/ && !/#<swpf>/ {
        if (Loop == 1) {
                Loop = 0;
                GotLine = 0;
                close(TmpFile)
        }
}
END {
    if (Loop == 1) close(TmpFile)
}
NAWK_FILE1_END
# Second awk program
cat << NAWK_FILE2_END > $nawk_file2
BEGIN {
```

```
    CurrentLine = 1
    TmpFileRoot = sprintf("$t/%s_SWP",FILENAME)
    Name = substr(FILENAME, 1, length(FILENAME)-3)
    SortInp = Name".sort"
    OutFile = Name".swp"; system("rm -f "OutFile);
    while ( (getline pair < SortInp) != 0 ) {
        split(pair,rec);
        NextLine = rec[1];
        NextInpFile = rec[2];
        while ( CurrentLine < NextLine ) {
            getline;
            print >> OutFile;
            ++CurrentLine;
        }
        system("cat " NextInpFile " >> " OutFile);
        system("rm " NextInpFile);
    }
}
{
    print >> OutFile;
}
END {
    system("rm " SortInp);
}
NAWK_FILE2_END
# compile all modules with -S given flags and modules specified
${TOOLROOT}/usr/bin/f77 -S $*
# for each module named on command line, process the output
set narg = $#argv
@ i = 1
while ($i <= $narg)
        if (($argv[$i]:e == f) || ($argv[$i]:e == F) || ($argv[$i]:e == c)) then
#           This guards against interpreting flags such as -WK,-inff=file.f
#           as files to compile.
if (-e $argv[$i]) then
                set s = $argv[$i]:r
                pr -t -n10 $argv[$i] > $s.pr
                nawk -f $nawk_file1 $s.s | sort -n > $s.sort
                nawk -f $nawk_file2 $s.pr
                /bin/rm $s.pr
            endif
        endif
        @ i = $i + 1
end
/bin/rm $nawk_file1
/bin/rm $nawk_file2
```

Shell Script ssruno

This script simplifies the run of a SpeedShop experiment.

Example C-6 SpeedShop Experiment Script ssruno

```
#!/bin/csh
# script to ssrun a program with designated output dir/filename.
# if no arguments, document usage
if (0 == $#argv) then
echo "$0 [-d output_dir] [-o output_file] [-ssrun_opts] prog_and_args"
   exit -1
endif
# initialize operands
set ssopts = ""
set otdir = "."
set otfile = ""
set proggy = ""
# collect -d, -o, and -ssrun options. Upon encountering name
# of program, break out of the loop, leaving $argv == prog_and_args
while ($#argv > 0)
   switch ($1)
   case "-o"
      setenv _SPEEDSHOP_OUTPUT_FILENAME $2
      set otfile = $2
      shift
      breaksw
   case "-d"
      setenv _SPEEDSHOP_OUTPUT_DIRECTORY $2
      set otdir = $2
      shift
      breaksw
   case "-*"
      set ssopts = ($ssopts $1)
      breaksw
   default
#     # get only tail, allowing ssrun /foo/bar/a.out
      set proggy = ${1}:t
      break
   endsw
   shift
end
# have to have seen a program
if ("X$proggy" == "X") then
   echo you must name a subject program
   exit -2
```

```
endif
# default the experiment type
if ("X$ssopts" == "X") then
   set ssopts = -usertime
endif
# run the experiment
echo ssrun $ssopts $argv....
ssrun $ssopts $argv
echo ...... ssrun ends.
# display all the output files with names starting $proggy
if ("X$otfile" == "X") then
#  # outfile not given, file is name.exptype.xpid
   ls -l $otdir/$proggy.*.?[0-9][0-9][0-9]*
else
#  # outfile given, file is name.xpid
   ls -l $otdir/$otfile.*
endif
```

Awk Script for Perfex Output

This script demonstrates one way to reduce and analyze the output of a *perfex* profile.

Example C-7 Awk Script to Analyze Output of perfex -a

```
# Reads output of perfex -a [-y]. Prints selected, reordered counters
# interpolating calculated ratios and percents.  Perfex runs of short
# programs often have zero values for some counts - allow for these.
BEGIN {
  maxline = 0 # track highest counter value seen
  mhz = 200 # assumed MHZ, adjust as needed
}
$0 ~ /^[ 123][0-9] / { # perfex data line
  lines[$1] = $0                # save the whole line
  counter[$1] = $NF             # save reported value
  if (maxline < $1) maxline = $1 # note high line# seen
}
END { # at end, print report
  if (maxline >=31)
  {
    print lines[0]
    seconds = counter[0]/(mhz*1000000)
    print "   " seconds " seconds elapsed at " mhz "MHZ"
    print lines[17]
    if (counter[17])
    {
```

```
  print "   " counter[0]/counter[17] " cycles/graduated instruction"
  print "   " (counter[17]/seconds)/1000000 " MIPS at 200MHZ"
  print lines[18]
  print lines[19]
  if (counter[18]*counter[19])
  {
    print "   " (counter[17]-counter[18])/counter[18] " instructions/load"
    print "   " (counter[17]-counter[19])/counter[19] " instructions/store"
    print "   " counter[18]/counter[19] " loads/store"
  }
  print lines[21]
  if (counter[21])
  {
    print "   " int((counter[21]/counter[17])*100) "% fp instructions"
  }
}
print lines[6]
print lines[24]
if (counter[6]*counter[24])
{
  print "   " int((counter[24]/counter[6])*100) "% branches mispredicted"
}
print lines[23]
if (counter[17]*counter[23])
{
  print "   " counter[17]/counter[23] " instructions/TLB miss"
}
print lines[9]
if (counter[17]*counter[9])
{
  print "   " counter[17]/counter[9] " instructions/i-L1 miss"
}
print lines[10]
print lines[11]
if (counter[17]*counter[10])
{
  print "   " counter[17]/counter[10] " instructions/i-L2 miss"
}
print lines[25]
if (counter[17]*counter[25])
{
  print "   " counter[17]/counter[25] " instructions/d-L1 miss"
}
print lines[26]
print lines[27]
```

```
    if (counter[17]*counter[25])
    {
      print "   " counter[17]/counter[26] " instructions/d-L2 miss"
    }
    smiss = counter[10]+counter[26]
    print "   " smiss " total L2 misses, " 128*smiss " bytes from memory"
    print "       " int(((128*smiss)/seconds)/1024) \
  " KB/sec memory bandwidth use at " mhz "MHZ"
    print lines[22]
    print lines[7]
    print lines[30]
    print lines[31]
    }
    else print "incomplete input"
}
```

Awk Script for Amdahl's Law Estimation

The script in Example C-8 can be run with the command *awk -f amdahl.awk*. Each line of input must be a list of numbers that represent execution times for one program using different numbers of CPUs. The *n*th number must be the execution time using *n* CPUs, T(*n*). Use 0 for an unknown time; however, at least the first and last numbers must be nonzero.

The script displays the calculated parallel fraction of the code, *p*, and the speedup and expected run time for various numbers of CPUs. Enter another line of times, or terminate the program with Ctrl+C.

Example C-8 Awk Script to Extrapolate Amdahl's Law from Measured Times

```
# amdahl.awk: an input line is a series of execution times
# T(1), T(2),...T(N) for a program run with 1, 2, ... N CPUs.
# Use 0 for an unknown time.  T(1) and T(N) must be nonzero.
# For example, after test with 1, 2, and 4 CPUs, you could enter
#           240 190 0 75
# to show those times, with 0 for the unknown time T(3).

{
   # save times T(n) in array t[]
   for (j=1;j<=NF;++j) t[j] = $j
   # calculate p, parallel fraction of code
   if (2==NF)
   {  # use simple formula for p given only T1, T2
```

```
      s2 = t[1]/t[2]
      p = 2*(s2-1)/s2
   }
   else
   {  # use general formula on the last 2 nonzero inputs
      for (m=NF-1; t[m]==0; --m) ;
      sm = t[1]/t[m]
      sn = t[1]/t[NF]
      invm = 1/m
      invn = 1/NF
      p = (sm - sn)/( sm*(1-invm) - sn*(1-invn) )
   }
   if (p<1)
   {
      printf("#CPUs   SpeedUp(n)    T(n)     p=%6.3g\n",p)
      npat = "%5d   %6.3g    %8.3g\n"
      # print the actual times as given and their speedups
      printf(npat,1,1.0,t[1])
      for (j=2;j<=NF;++j)
      {
         if (t[j]) printf(npat,j,t[1]/t[j],t[j])
      }
      # extrapolate using amdahl's law based on calculated p
      # first, for CPUs one at a time to 7
      for (j=NF+1;j<8;++j)
      {
         sj = 1/((p/j)+(1-p))
         printf(npat,j,sj,t[1]/sj)
      }
      # then 8, 16, 32, 64 and 128
      for (j=8;j<=128;j=j+j)
      {
        sj = 1/((p/j)+(1-p))
        if (j>NF) printf(npat,j,sj,t[1]/sj)
      }
   }
   else
   {
      printf("p=%6.3g, hyperlinear speedup\n",p)
      printf("Enter a list of times for more more than %d CPUs\n\n",NF)

   }
}
```

Page Address Routine va2pa()

This routine allows a program to pass the address of any variable, and recover the physical memory address of the page containing the variable. It can be used to investigate memory distribution effectiveness.

You can translate a virtual address to a node number with the following macro, which calls **va2pa()**.

```
#define VADR2NODE(A) ((int) (va2pa(A) >> 32))
```

You can retrieve the CPU number instead of a node number using this macro:

```
#define VADR2CPU(A) ((int) (va2pa(A) >> 16))
```

Example C-9 Routine va2pa() Returns the Physical Page of a Virtual Address

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/syssgi.h>
__uint64_t
va2pa( void *va)
{
   __uint64_t   pa;
   __uint32_t   pfn;
   int          status;
   static int   lpgsz, pgsz = -1;
   if (pgsz < 0) { /* first time: log2(pagesize) */
      int itmp;
      pgsz = itmp = getpagesize();
      for (lpgsz=0; itmp>1; itmp>>=1, lpgsz++);
   }
   if ((status = syssgi(SGI_PHYSP,va,&pfn)) != 0) {
      perror("Virtual to physical mapping failed");
      exit(1);
   }
   pa = (((__uint64_t) pfn << lpgsz) | ((__uint64_t) va & (pgsz-1)));
   return (pa);
}
```

CPU Clock Rate Routine cpuclock()

This routine gets the clock rate in megahertz of the first CPU listed in the hardware inventory, and returns it as an integer. You can use this number to convert an elapsed time into a count of CPU cycles.

Example C-10 Routine cpuclock() Gets the Clock Speed from the Hardware Inventory

```
/* ============================================================
|| Return CPU clock rate in megahertz, by the rather
|| byzantine method of scanning the hardware inventory
*/
#include <invent.h>
#DEFINE DFLT_MHZ 195 /* return if any error */
int cpuclock(void)
{
        inventory_t *p_inv;
        if (setinvent()) return DFLT_MHZ;
        for(p_inv = getinvent(); (p_inv); p_inv = getinvent())
        {
                if (  (p_inv->inv_class == INV_PROCESSOR)
                        &&(p_inv->inv_type == INV_CPUBOARD) )
                break;
        }
        endinvent();
        if (p_inv)
                return p_inv->inv_controller;
        else
                return DFLT_MHZ;
}
```

Glossary

This short glossary contains only terms related to performance tuning, as used in this guide. For a full glossary of Silicon Graphics technical terms, see the *Silicon Graphics Glossary of Terms*, 007-1859-xxx.

Amdahl's Law

A mathematical approximation (not really a law) stating the ideal speedup that should result when a program is executed in parallel: S(n) = 1 / ((p / n) + (1 - p)), where n is the number of CPUs applied and p is the fraction of the program code that can be executed by parallel threads. When p is small (much of the program executes serially), S(n) is near 1.0, in other words, there is no speedup. When p is near 1.0 (most of the program can be executed in parallel), S(n) is near $1/n$, in other words, linear speedup.

bandwidth

The amount of data transferred per second over a given path. Common types are:

- Theoretical bandwidth, the maximum units per second under perfect conditions
- Available bandwidth, units per second of useful data after subtracting protocol overhead and turnaround delays in normal traffic situations
- Bisection bandwidth, the maximum available bandwidth between two arbitrarily-chosen points in the system allowing for bottlenecks.

See also *latency*.

basic block

A sequence of program statements that contains no labels and no branches. A basic block can only be executed completely (because it contains no labels, it cannot be entered other than at the top; because it contains no branches, it cannot be exited before the end), and in sequence (no internal labels or branches). Any program can be decomposed into a sequence of basic blocks. The compiler optimizes units of basic blocks. An ideal *profile* counts the use of each basic block.

branch prediction

See *speculative execution*.

cache

In common language, a hiding-place for valuables, provisions, and so on. In computing, a separate memory where the most recently used items are kept in anticipation of their being needed again. The MIPS R10000 CPU contains an on-chip cache for instructions and another for data (the *primary cache* or L1 cache). The SN0 *node* provides a *secondary cache* (or L2 cache) of either 1 MB or 4 MB, organized as 128-byte units called *cache line*s.

cache blocking

An optimization technique that modifies a program so that it works on blocks of data that fit in *cache*, and deals completely with each block before going on to another.

cache coherency

In a multiprocessor, each CPU has its own cache; hence any memory item can appear in multiple copies among the various caches. Cache coherency hardware ensures that all cached copies are identical to memory. The basic method is to recognize when one CPU modifies memory, and to automatically invalidate any other copies of the changed data. See *cache*, *directory*, *snoopy cache*.

cache contention

When multiple CPUs update the same *cache line*, each has to become the exclusive owner of that line in turn. This can drastically slow execution. Not the same phenomenon as *cache thrashing*, but caused by *memory contention* or by *false sharing*.

cache directory

A design for *cache coherency* used in the SN0 architecture. Each *secondary cache cache line* of 128 bytes is augmented with a set of directory bits, one bit for each CPU that could have a cached copy of that line. When a CPU accesses a cache line, its bit is set in the directory of the node that owns that line. When a CPU modifies memory, all CPUs whose directory bits are set in the directory of the associated line are notified to invalidate their copies of the line.

cache line

The unit of memory fetched and stored in a *cache*. Cache line size is a function of the hardware. In the MIPS R10000 CPU, the *primary cache* line is 32 bytes. In the SN0 (and most other Silicon Graphics systems) the *secondary cache* line is 128 bytes. See *cache*.

cache miss

When the CPU looks for a particular memory word in a *cache* and does not find it. Execution of the current instruction has to be delayed until the *cache line* has been loaded from memory. A *superscalar* CPU can continue to execute other instructions out of order while waiting.

cache thrashing

Cache lines are assigned locations in cache based on the bits of their physical address. It is possible for cache lines from unrelated parts of memory to contend for the same location in cache. When a program cycles through multiple arrays that happen to have similar base addresses, it is possible for every fetched cache line to displace another recently fetched line. The SN0 uses a "two-way set-associative" cache that can deal with alternating references to two arrays with similar address bits, but it can be defeated by a loop that refers to three or more arrays. This "thrashing" effectively negates the cache, reducing access to memory speed. It is prevented by padding arrays to force their starting addresses to differ in the least-significant bits. See also *false sharing*, which is quite different.

cache tiling

See *cache blocking*.

counter

See *hardware counter.*

dependency

When one instruction or program statement requires the result of another as its input, and so depends on the completion of the other. A loop contains a dependency when one iteration of the loop requires the result of a prior iteration; such loops cannot be parallelized. In a *superscalar* CPU, an instruction cannot complete until its dependencies are satisfied by the completion of the instructions that prepare its operand values.

directory

See *cache directory.*

dynamic page migration

page migration that is initiated by the operating system, based on hardware counters in the *hub* that show which *node*s are using any given page of memory. Dynamic page migration can be turned on for the whole system (not recommended) or for a specific program by running it under *dplace*.

false sharing

When two or more variables that are not logically related happen to fall into the same *cache line*, and one variable is updated, a reference to any of the other variables cause the cache line to be fetched anew from memory even though the data in those variables has not changed. In effect, false sharing is unintentional *memory contention*. False sharing is eliminated by putting volatile variables in cache lines that are shared only by variables that are logically associated with them and that are updated at the same time.

first touch allocation

The default rule for IRIX memory allocation in the SN0: new memory pages are allocated in the *node* where they are requested, or in the nearest node to it. See also *round robin allocation*.

gather-scatter

A compiler optimization technique for improving the speed of a loop that iterates over an array, testing each element and operating on only selected elements. Because such a loop contains a conditional branch, *software pipelining* is less effective in it. The loop is converted into two loops: one that tests all elements, collecting the indexes of the target elements in a scratch array; and a second loop that iterates over the scratch array and operates on the selected elements. The second loop, because it contains no conditional branch, can be more effectively optimized.

graduated instruction

In a *superscalar* CPU, instructions are executed out of the order in which they were written. In order to ensure that the program state changes only in ways that the programmer expects, any instruction that completes early is delayed until the instructions that logically preceded it in the program text have also completed. Then the instruction "graduates" (as a student graduates from school) and its effect on the program state is made permanent. Under *speculative execution*, some instructions are executed and then discarded, never to graduate. An R10000 CPU *hardware counter* can count instructions issued and graduated.

hardware counter

In the MIPS R10000 CPU, one of two registers that can be programmed to count events such as *cache miss*, *graduated instruction*, and so on. Used generating a sampled *profile*.

hub

In the SN0 architecture, the custom circuit that manages memory access in a *node*, directing the data flowing between attached I/O devices and CPU chips to memory in the node and through a *router* to other nodes.

hypercube

In *topology*, the four-dimensional figure whose faces are cubes of identical size (as the cube is the 3D figure whose faces are squares of identical size). In an SN0 system with 32 or more CPUs, the CPU *node*s are connected by data paths that follow the edges of a hypercube.

inlining

A compiler optimization technique in which the call to a procedure is replaced by a copy of the body of the procedure, with the actual parameters substituted for the parameter variables. Inlining has two immediate effects: to eliminate the overhead of calling the procedure, and to increase the size of the code. It also has the side-effect of giving the compiler more statements to practice optimization upon. For one example among many, when one of the actual parameter values is a constant, it may be possible to eliminate many of the conditional statements from the inserted code because, with that actual data, they are unreachable.

interprocedural analysis (IPA)

A compiler optimization technique. Normally a compiler analyzes one procedure at a time in isolation, making the most conservative assumptions about the possible range of argument values it might receive and about the actions of the procedures that it calls. Using IPA, the compiler examines all calls to each procedure to determine the actual ranges of arguments, and to determine whether or not a called procedure can ever modify shared data. The information gained from IPA permits the compiler to do more aggressive optimization, but to get it, the entire program text must be available at once. The current Silicon Graphics compilers implement IPA by performing only syntax analysis at "compile" time, and performing the actual compilation at "link" time, when all modules are brought together.

kthread

The IRIX kernel, since release 6.4, is structured as a set of independent, interruptible, lightweight threads. These "kthreads" (kernel threads) are used to implement system functions and to handle interrupts and such background tasks as disk buffering. Some kthreads are used to execute the code of threads in user-level programs. However, a kthread and a *thread* are quite different in their abilities and execution environment.

latency

Regarding memory or a communications path, the time elapsed from the instant a request is issued until the response begins to arrive. When communications consists of many short exchanges, latency dominates the data rate. When messages are long enough that the transmission time of a message is larger than latency time, *bandwidth* dominates. Regarding a CPU, latency is the time from loading an instruction to completing it.

libfastm

A highly optimized version of the standard Fortran math library.

loop fission

A compiler optimization technique, the opposite of *loop fusion*, in which a loop whose body is too large for effective optimization or *software pipelining* is automatically split into two loops. May also be done when the loop body contains inner loops such that, after the fission, *loop interchange* can be performed.

loop fusion

A compiler optimization technique in which two adjacent loops that have the same induction variable range are merged. The statements in the loop bodies are put together within a single loop structure. Benefits include halving the time spent on loop overhead; putting more statements in the *basic block* to give the compiler more material for *software pipelining*; most important, using cache data more effectively. See also *loop peeling*.

loop interchange

A compiler optimization technique in which the loop variables of inner and outer loops are interchanged. This is done either to reduce the *stride* of the loop, to improve its cache use, or to make *cache blocking* or *loop unrolling* possible.

loop nest optimization

A compiler optimization technique in which the statements in nested loops are transformed to give better cache use or shorter instruction sequences.

loop peeling

A compiler optimization technique in which some of the first (or last) iterations of a loop are made into separate statements preceding (following) the loop proper. This is done in order to permit *loop fusion* with an adjacent loop that uses a smaller range of indexes.

loop unrolling

A compiler optimization technique in which the body of a loop is replicated several times, and the increment of the loop variable is adjusted to match (for example, instead of using array element a(i) in one statement, the loop has four statements using elements a(i), a(i+1), a(i+2), and a(i+3), and then increments i by 4). This divides the loop overhead by the amount of unrolling, but more importantly gives the compiler more statements in the *basic block* to optimize, for example making *software pipelining* easier. See also *loop nest optimization*.

memory contention

When the code in two or more CPUs repeatedly updates the same memory locations, execution is slowed due to *cache contention*.

memory locality

The degree to which data is located close to the CPU (or CPUs) that access it. In the SN0 architecture, there is a (small) time penalty for access to memory on a different *node*, so the operating system applies many strategies to keep data close to the CPUs that use it.

memory locality domain (MLD)

An abstract representation of physical memory, used within the IRIX operating system. An MLD is a source for memory allocation having a prescribed type of *memory locality,* for example, the two nodes in one corner of a *hypercube* can yield memory that is close to either node.

Million Floating Point Operations Per Second (MFLOPS)

A measure of the speed of execution of a program. A program contains many non-floating-point instructions, so the MFLOPS achieved on the same hardware, varying the compiler options, is an indication of how effectively the compiler optimized the code. Often, MFLOPS is reported through an approximate calculation, based on counting the number of arithmetic operations in the source code for a crucial loop body and multiplying by the number of iterations. However, the *hardware counter* of the R10000 CPU can count actual floating point *graduated instruction*s, giving an absolute measure of MFLOPS.

Million Instructions Per Second (MIPS)

A measure of the speed of execution of a CPU. A *superscalar* CPU such as the MIPS R10000 CPU can normally finish 1.5 instructions per clock cycle, or about 300 MIPS for a CPU with a 200MHZ clock.

node

The basic building-block of the SN0 architecture, a single board carrying two R10000 CPUs, the *secondary cache* for each, some memory, and the *hub* chip that controls data flow to and from memory.

Non-uniform Memory Access (NUMA)

A computer memory design that has a single physical address space, but in which parts of memory take longer to access than other parts. In the SN0, memory in the local *node* is accessed fastest; memory in another node takes longer by about 100 nanoseconds per *router*.

optimization

Finding the solution that is the best fit to the available resources. See *tuning*.

page migration

Moving a page of memory data from one *node* to another one that is closer to the CPU that is using the data in the page. Page migration can be explicitly programmed using Fortran directives or runtime calls on the *dplace* program. See also *dynamic page migration*.

parallelization

Causing a program to execute as more than one *thread* operating in parallel on the same data. Often specified by using C pragmas or Fortran directives to make loops run in parallel, or by writing explicitly parallel code that starts multiple processes or *pthread*s.

policy module (PM)

The IRIX kernel data structure type that specifies one *memory locality domain (MLD)* and other rules regarding memory allocation from that domain.

prefetch

An instruction that notifies the CPU that a particular *cache line* will be needed by a following instruction, permitting memory fetch to overlap execution. The prefetch instruction is simply a load that does not use its operand. Current Silicon Graphics compilers can generate prefetch instructions automatically.

primary cache

The on-chip cache memory, closest to the execution units and with smallest latency. In the MIPS R10000 CPU, primary cache is 32KB organized in 32-byte units. Also called L1 (level-one) cache. See *cache*, *secondary cache*.

profile

To analyze the execution of a program by counting the number of times it executes any given statement. An *ideal* profile is done by modifying the executable so that it contains instructions to count the entry to each *basic block*. A *sampled* profile is done by executing the unmodified executable under a monitor that interrupts it at regular intervals and records the point of execution at each interruption. A sampled profile emphasizes different kinds of behavior depending on the chosen sampling time base.

pseudo-prefetch

To order data references in a loop so that a harmless reference to the next cache line is executed well in advance of the use of the data from that cache line. A manual version of *prefetch* often used to hide the latency of loading the *primary cache*.

pthread

A *thread* of a user program, created and controlled using the POSIX 1003.1c standard interface. An alternative to *parallelization* using multiple IRIX processes. Not related in any way to *kthread*.

round robin allocation

An alternative to *first touch allocation* in which new pages are distributed in rotation to each *node*. This prevents the *hub* chip in one node from having to serve all requests for memory from one program.

router

In the SN0, the custom board that routes memory access between *node*s.

scalability

The quality of scaling, or increasing in proportion, in a regular fashion based on the availability of resources. In hardware, increasing in performance as a smooth function of increasingly complex circuitry; or conversely, increasing in cost as a smooth function of capacity. In software, improving execution time as a smooth function of available CPUs and memory (see *Amdahl's Law*).

secondary cache

Cache external to the CPU chip; also called L2 (level-two) cache. See *cache, primary cache*.

snoopy cache

A design for *cache coherency* used in the CHALLENGE and Onyx systems. Each CPU board monitors all bus traffic. When it observes a memory write, it invalidates any cached copy it might have of the same memory. This is the simplest way to implement cache coherency, but it depends on having all CPUs on a single memory bus, which is not the case in the SN0.

software pipelining

A compiler technique in which the compiler generates sequences of instructions that are carefully tailored to take maximal advantage of the multiple execution units of a *superscalar* CPU.

speculative execution

A *superscalar* CPU can decode and begin processing multiple instructions in parallel. When it decodes a branch instruction, the condition tested by the branch may not yet be known, because the instruction that prepares that condition is still executing or perhaps is suspended pending yet another unknown result. Rather than suspending execution pending completion of the branch input operand, the CPU predicts the direction the branch will go, and continues to decode and execute instructions along that path. However, it does so in a way that any changes in program state that result can be retracted. If the prediction turns out to be wrong, the results of the speculatively executed instructions are discarded. Experience with the branch prediction logic of the MIPS CPUs shows that branch prediction is typically correct well over 90% of the time. Branch mispredictions can be counted in a *hardware counter*.

stencil

Name for a family of similar algorithms that combine the value from a cell in an input array with the values in its neighbor cells to make a new value for the same cell of an output array. A 2D stencil operates on the 9-cell neighborhood comprising a(i,j) and its eight neighbors (a(i-i,j), a(i+1,j), and so on). A 3D stencil operates on the 27-cell neighborhood around a(i,j,k). Conway's Game of Life and other cellular automata are usually implemented using a stencil algorithm, as are many graphics transformations such as smoothing and edge-finding. Stencil algorithms are both CPU-intensive and memory-intensive, and are challenging subjects for compiler optimizations.

stride

The distance between the memory addresses of array elements that are touched in a loop. A stride-one loop touches successive array elements, and hence scans memory consecutively. This uses *cache* memory most efficiently because all of a *cache line* is used before the next cache line is fetched, and the loop never returns to a cache line after using it. Strides greater than one are less efficient in memory use, but are easy to create accidentally given Fortran array semantics. The compiler can sometimes use *loop nest optimization* or *loop interchange* to shorten the stride.

superlinear speedup

When adding more CPUs to parallel execution of a program, the program normally speeds up in conformity with *Amdahl's Law*. However, when adding CPUs accidentally relieves some other bottleneck, the speedup can exceed the number of CPUs added. This is superlinear speedup: improvement out of proportion to the hardware added.

superscalar

In a RISC CPU, the ability to execute more than one instruction per clock cycle, achieved by having multiple parallel execution units. The MIPS R10000 CPU routinely achieves instruction rates of 1.5 to 2 times its clock rate. It uses five, independent execution units, and is able to execute instructions out of order, so that it can complete some instructions while waiting for the memory operands of others to arrive. See also *speculative execution* and *software pipelining*.

thread

Loosely, any independent execution of program code. A thread is most commonly represented by an IRIX process; or by a POSIX thread using the *pthread* library. When discussing *parallelization* of a program, it is usually assumed that there are as many CPUs as the program has threads, so all threads can execute concurrently. This is not assumed in the general case; the threads of a multithreaded program are dispatched by IRIX as CPUs are available, and some may run while others wait. See also *kthread*.

topology

The ways in which the SN0 nodes are connected in general (see also *hypercube*); but in particular the relationship between the nodes in which the various threads of a parallel program are executed. Typical program topologies are cluster (which minimizes the distance between nodes), cube, and hypercube.

translation lookaside buffer (TLB)

A table within the CPU that associates virtual addresses to physical page addresses. The TLB has limited size, but all entries are searched in parallel, simultaneously with the decoding of the instruction (hence "lookaside"). When the operand address of an instruction is found in the TLB, the CPU can present the needed physical address to the memory without delay. (The TLB is in essence a *cache* for page table entries.) When the TLB lookup fails, the CPU traps to an OS routine that locates the desired virtual page in a large, in-memory page table that defines the virtual address space of the process. When that lookup succeeds, the OS loads one entry of the TLB to address the page, and resumes execution. When that lookup fails, the process has suffered a page fault.

tuning

Finding the executable version of a program that uses the least of a critical resource on a given hardware platform. The critical resource is usually time; the tuned program runs fastest. However, tuning for least memory or I/O use is also possible. Tuning is done by: altering the program's source code to use a different algorithm; by compiling the program with a different compiler or with different compiler options to get a more highly optimized executable; or changing the numbers of CPUs or placement of memory at execution time.

vector intrinsic

A Fortran (usually) math function that is implemented using hardware vector operations. The compiler can substitute vector intrinsic calls automatically.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

X

Z

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3430-002.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389