Forma: A Framework for Safe Automatic Array Reshaping

PENG ZHAO,
University of Alberta
SHIMIN CUI, YAOQING GAO, RAÚL SILVERA
IBM Toronto Software Laboratory,
and
JOSÉ NELSON AMARAL
University of Alberta

This paper presents *Forma*, a practical, safe, and automatic data reshaping framework that reorganizes arrays to improve data locality. *Forma* splits large aggregated data types into smaller ones to improve data locality. Arrays of these large data types are then replaced by multiple arrays of the smaller types. These new arrays form natural data streams that have smaller memory footprints, better locality, and are more suitable for hardware stream prefetching. *Forma* consists of a field-sensitive alias analyzer, a data type checker, a portable structure reshaping planner, and an array reshaper. An extensive experimental study compares different data reshaping strategies in two dimensions: (1) how the data structure is split into smaller ones (*maximal partition* × *frequency-based partition* × *affinity-based partition*); and (2) how partitioned arrays are linked to preserve program semantics (*address arithmetic-based reshaping* × *pointer-based reshaping*). This study exposes important characteristics of array reshaping. First, a practical data reshaper needs not only an inter-procedural analysis but also a data type checker to make sure that array reshaping is safe. Second, the performance improvement due to array reshaping can be dramatic: standard benchmarks can run up to 2.1 times faster after array reshaping. Array reshaping may also result in some performance degradation for certain benchmarks. An extensive micro-architecture-level performance study identifies the causes for this degradation. Third, the seemingly naive maximal partition achieves best or close-to-best performance. This paper presents an analysis that explains this surprising result. Finally, address-arithmetic-based reshaping always performs better than its pointer-based counterpart.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; *Code generation*; *Optimization*; D.3.3 [**Programming Languages**]: Language Constructs and Features; E.1 [**Data Structures**]: Arrays

General Terms: Algorithms, Performance, Memory Hierarchy

Additional Key Words and Phrases: Arrays, Data Structure, Reference Analysis

---

Author's address: P. Zhao and J. N. Amaral, Department of Computing Science, University of Alberta, Edmonton, Canada.

1. INTRODUCTION

Fast advances in semiconductor fabrication, architectural innovation and exploitation of instruction-level parallelism (ILP) ensures that the performance potential of modern processors continues to increase at a substantial speed. However, the performance of a computer system is not solely determined by the processor. To maintain a high utilization of its functional units, a fast processor must be efficiently fed with instructions and data by a memory subsystem. Unfortunately, the speed of memory continues to lag behind that of processors. In recent decades, the clock rate of processors has approximately doubled every three years while the DRAM access speed only increased about 50% [**?**].

The solution to overcoming this increasingly insurmountable "memory wall" is to improve caching systems. Because of fabrication constraints, power consumption and economics, the cache size is often very limited when compared with main memory. Hence, efficient utilization of cache is crucial for performance. However, because cache is transparent to application programmers, programs are often written without taking cache efficiency into account. The typical programmer designs data structures in a semantic-oriented fashion. Such structures are usually easy to read and understand, but often lead to suboptimal performance at run time. A compiler can analyze the memory reference pattern of a program and devise a new layout that increases locality of references and thus improves the efficiency of the memory system. Similar to traditional code transformations, these data transformations must be transparent to the programmer, performed automatically, and safe.

This paper describes *Forma*[1], a framework that improves the data cache efficiency of arrays of aggregate data structures. An extension of this framework to handle linked data structures (LDS) is discussed in Section **??**. Aggregate data types (such as `structs` and `classes`) are used to model objects in imperative programming languages. When the object modeled has many features, the aggregate data type becomes very large. Arrays that contain large data structures with many fields occur frequently in contemporary programs. Because programmers often arrange the fields in a data type in a way that is semantically meaningful, there is a tension between software engineering and performance engineering that presents itself in two ways. First, the frequency of access to fields in the same data type may vary significantly, with *hot fields* accessed very frequently and *cold fields* seldom referenced. Placing fields with very different access frequencies together in memory hurts performance because the cold fields pollute the data cache and waste memory bandwidth. Second, the runtime data access pattern might not be consistent with the access frequency distribution. In other words, hot fields are not necessarily accessed together. The gap between software engineering and performance engineering can be bridged by reshaping data at compile time. The compiler can split a large data structure into two or more smaller ones that better capture data locality. Correspondingly, an array of large data structures is partitioned into two or more arrays of smaller data structures. For iterations that only manipulate certain fields of the array, data reshaping can significantly improve data locality and reduce the

[1] *Forma* is a Latin, and Portuguese, word for "shape".

memory footprint, resulting in better data cache efficiency.

The main contributions of this paper are:

—*Forma*, a practical data reshaping framework that can be used to automatically analyze and transform real C/C++ programs. *Forma* consists of a data shape analysis, including both alias analysis and data type analysis, structure partition planning and array reshaping transformation. *Forma* has been integrated into the IBM® XL C/C++ V7.0 compiler.

—A set of safety-checking rules to ensure that the compiler's data reshaping plan is safe in programs that are written in type-unsafe languages such as C and C++.

—An empirical study of two orthogonal reshaping decisions: frequency-based object partition × affinity-based object partition × maximum object partition; and address-arithmetic-based × pointer-based array splitting. Some important but subtle insights on data reshaping are exposed by a thorough analysis and empirical study.

The rest of the paper is organized as follows. Section **??** introduces *Forma* and two design dimensions of data reshaping. Then the performance of the different data reshaping approaches is studied in Section **??**. Section **??** discusses related work on data cache optimization. Finally, conclusions and a discussion on extensions of *Forma* to handle data topology other than arrays appear in Section **??**.

2. DATA RESHAPING

2.1 Overview

Previous research work on field placement and data structure splitting appears in [**?**; **?**; **?**; **?**]. However, some of these studies use error-prone human-inspection of C applications to make sure that the transformation is safe [**?**; **?**]. Applying a type-safe-oriented optimization to a type-unsafe language without a proper safety assurance mechanism is unacceptable in production compilers. Rabbah *et al.* use field-insensitive Steensgaard style analysis to find the alias sets that need to be updated upon data splitting [**?**; **?**]. However, alias analysis alone cannot guarantee the transformation safety. Also, as our work illustrates, field-sensitivity in alias analysis is crucial to uncovering important array reshaping opportunities in many integer benchmarks.

This paper describes *Forma*, a complete framework that performs ***automatic*** and ***safe*** data reshaping on type-unsafe programming languages such as C/C++. In contrast with existing work, *Forma* is fully automatic, safety-guaranteed, and more aggressive on alias analysis in terms of field sensitivity than previous work. *Forma* was designed and implemented in the IBM®XL C/C++ V7.0 compiler suite.

As illustrated in Figure **??**, *Forma* consists of three components: a data shape analyzer, a structure partition planner, and a whole-program data reshaper. *Forma* requires two passes through the entire program: a data shape analysis pass and a data reshaping pass. The data shape analysis pass includes alias analysis and data type analysis. In this pass, a storage shape graph [**?**] is constructed to model the aliasing relationships in the program. *Forma* also examines whether the data types[2] of the members in an alias set are consistent throughout the entire program.

[2] In this research, we use *data type*, *data shape* and *data view* interchangeably.

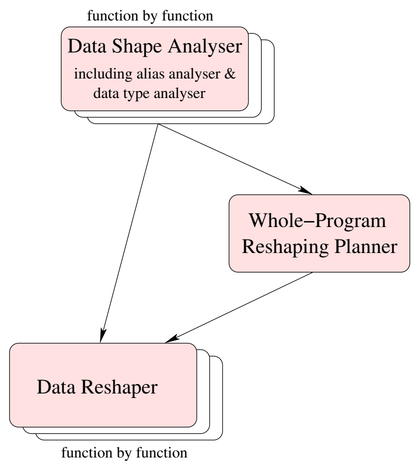


Fig. 1. The *Forma* ata reshaping framework.

If array reshaping is deemed safe and beneficial, at the end of the first pass *Forma* creates a partition plan for the composite data structure. Then the data reshaping pass adjusts the memory accesses according to this plan.

2.2 Data Shape Analysis

An inter-procedural data shape analysis generates information about alias relationships and data shape consistency. Alias information provides a conservative approximation of sets of data objects that potentially reside in the same memory location. Data shape includes *structural shape* and *array shape*. Structural shape describes the field-level view of a singular data object, *i.e.* the number of byte-level fields, the offset and length of each byte-level field. Array shape is the view of an array. It consists of the number of dimensions and the stride for each dimension of the array. If the view of an array throughout the program is not consistent, the compiler must be conservative and give up data reshaping optimization on the array.

2.2.1 *Inter-procedural Alias Analysis.* In a program written in a pointer-rich language, such as C and C++, reshaping a data object might impact the whole program because of aliasing relationships. Therefore, a compiler needs to modify all the affected references when it reshapes a data object. *Forma* focuses on reshaping arrays. If an array is to be reshaped, all the references to the array area need to be modified accordingly. This comprehensive transformation requires *Forma* to conduct an inter-procedural alias analysis to collect all the pointers pointing to the array area.

Inter-procedural alias analysis techniques differ in flow sensitivity, context sensitivity, field sensitivity, and so on. A good survey of these techniques can be found in [**?**]. *Forma* implements a Steensgaard style alias analysis [**?**] that has the following characteristics:

—It is flow-insensitive. It conservatively assumes that all the statements in a pro-

```
char *pc;  int i,j;
struct A {char *f1; struct A *f2;} *p1, *p2;
p1 = malloc (DIM * sizeof (A));
p2 = &(p1[j]);
p1[j].f1 = pc;
p1[i-1].f2 = &(p1[i]);
// pc = (char*) p1;
```

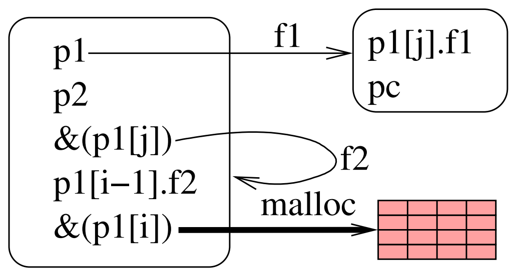


Fig. 2. Field-sensitive Steensgaard alias analysis.

cedure will be executed in arbitrary order. Thus the control flow information is irrelevant.

—It is context-insensitive. It does not differentiate the alias relationship created in different calling contexts.

—It is unification-based. Whenever a pointer assignment is met in the analysis, the alias sets represented by the right-hand side pointer and by the left-hand side pointer are merged. The points-to sets of these pointers are also merged. This assumption eliminates the iteration over the control flow graph (CFG) that is required by inclusion-based alias analysis. Therefore, the analysis can be completed with a single pass through the entire program.

Steensgaard's alias analysis was selected for *Forma* for two reasons. First, it is much simpler and faster than other alias analysis [**?**]. Second, the precision of alias relationships provided by Steensgaard's alias analysis is sufficient for data reshaping. To reshape an array, it is sufficient to know all the data accesses to the array. There is no need to know precise aliasing relationships such as whether two pointers actually point to exactly the same element of the array. The extra precision provided by flow-sensitivity, context-sensitivity, and directionality in a more complex and more expensive alias analysis would be redundant.

Field-sensitivity is important for data reshaping because large data structures, which are amenable to data reshaping, often contain pointers pointing to different data types. A field-sensitive alias analysis is necessary to distinguish the alias relationships among different fields or between a field and its *host object*. The host object is the object that contains the field. For instance, Figure **??** depicts a code segment and the corresponding storage shape graph generated by a field-sensitive Steensgaard alias analyzer. The pointer manipulation in the example is pretty common in practice. The bold arrow represents the allocation site in the program. Thin arrows represent the points-to relationships among expressions. In the example, the points-to set is divided into two categories according to the fields.

That is, field `f1` and field `f2` should never reference the same address. The field-sensitivity is crucial for this code segment. In a field-insensitive analysis, access to any field is regarded as an access to the entire host object. Therefore, `pc` and `p1[j].f1` would be in the same alias set as `p1`. *Forma* would conclude that the `p1`'s alias set could be either an `A`-typed pointer or a `char` pointer. Then *Forma* would have to give up the data reshaping of `p1`'s alias set because the data types of the members in the alias set would be regarded as inconsistent. In *Forma*, field-sensitivity is achieved by a technique similar to those in [?; ?].

2.2.2 *Reshaping Safety.* C/C++ are weakly type-checked programming languages. This means that even extensive type checking in a compiler front-end cannot detect all unsafe operations. Some of the type loopholes were intentionally included in these languages to enable performance-efficient and code-convenient implementations of system software. For example, let's examine the commented statement in Figure **??**. The address of the allocated memory block is cast to type *char* and assigned to pointer `pc`. This casting is very common in the implementation of low-level communication libraries: a buffer is filled with an array of high-level data objects and then streamlined and sent to the lower transferring layers. Reshaping on data that has incompatible type is dangerous and is strictly avoided in our work.

In *Forma* two intrinsic data types are compatible if their data lengths are identical. This is a more relaxed definition when compared with that from [?]. Two aggregated data structures are compatible if (1) they have the same number of byte-level fields[3], (2) corresponding fields have the same offset and length, and (3) their addresses are either identical or don't overlap with each other. Two arrays have compatible types if (1) their element types are compatible, (2) they have the same dimensions, and (3) the strides of corresponding dimensions are also identical. Two pointers are of compatible types if and only if the data they point to have compatible types.

Forma conducts type-compatibility checks to avoid dangerous data reshaping. *Forma* does not attempt to improve the type-safety of a program. On the other hand, the data reshaping transformation must be carefully implemented to avoid introducing new, potentially unsafe, runtime type errors to the program. Safety is achieved by integrating a type compatibility analysis with the inter-procedural alias analysis: the types of the members in an alias set must be compatible throughout the application. The inter-procedural alias analysis keeps track of the types of each alias set. Once a type incompatibility is found, the alias set is abandoned for reshaping analysis.

The compatibility rule is necessary to ensure the safety of the transformation. It requires that all the access patterns in an alias set be verifiably consistent. For example, if a pointer is passed to system libraries and the compiler cannot examine the access pattern in the libraries, the alias set represented by the pointer must be abandoned. Fortunately, a large set of programs satisfy these seemingly restrictive conditions [?; ?].

[3]We assume the bit fields are converted to byte-level fields.

Section **??**, introduces two array splitting strategies. One of these strategies, address-arithmetic-based splitting, requires an extra restriction to ensure its safety.

2.3 Structure Partition Plan and Array Reshaping

If an array is deemed safe for reshaping, *Forma* makes a partition plan for the aggregated data structure of the elements of the array. The shape of the original array will change to satisfy the data structure partition. Then *Forma* transforms all the accesses to the array to make them compatible with the new array shape and the new aggregated data structures.

This section describes three structure partition planners. Two approaches to reshape an array according to the structure partition plan are discussed in Section **??**. Section **??** presents a detailed empirical performance study of reshaping and splitting.

2.3.1 *Structure Partition Planner.* A structure partition plan determines how the fields in the original data structure should be reorganized into new data structures. For example, consider the four-field data structure, O_{orig}, in Figure **??**. The fields of O_{orig} are numbered from F0 to F3. Each rectangle represents a field, and rectangles with the same filling pattern are fields that are always accessed together. In this example, field F0 and field F3 have high access affinity, while field F2 is always accessed alone. Assume that field F1 is very cold and the other fields are hot. In Figure **??**, the fields in O_{orig} are reordered and split into three smaller structures: O_{base}, O_{sat_1} and O_{sat_2}. Each new structure has its own length and offset in the reorganized data structure. O_{base} is the *base structure* or *base object* and starts from offset 0. The other new data structures, O_{sat_1} and O_{sat_2}, are the *satellite structures* and are placed immediately after O_{base}.

Array reshaping is based on a data structure partition plan. After array reshaping, different partitions of the same object might be placed far apart from each other. Figure **??** shows an array, A_{orig}, of four elements of type O_{orig}. Figures **??**, **??**, and **??** show the effects of different structure partition plans on A_{orig}. In these figures, each rectangle is a field of O_{orig} and Fij represents the jth field in the ith element of the original array. From the array point of view, the original array A_{orig} is split into one base array A_{base}, which holds the base objects, and one or more satellite arrays. For instance, each element might be split into three new objects according to the partition plan shown in Figure **??**. Correspondingly, A_{orig} might be split into three arrays, as shown in Figure **??**. The first two rows in Figure **??** are the new base array A_{base} and the other two rows are the new satellite arrays. How the satellite fields are accessed depends on the array-splitting approach, as described in Section **??**. To correctly reference satellite fields that are placed in satellite objects, extra address calculations are needed. Therefore, if hot fields are split into satellite objects, there might be a substantial increase in the number of instructions executed to compute the address of hot satellite fields.

Forma implements the following three data structure partition strategies.

Affinity-Based Planner (ABP). Some of the hot fields in a data structure have higher *access affinity* than others, *i.e.*, they tend to be referenced together. Therefore, a natural solution is to split a data structure according to access affinity: the fields in a data structure are clustered by reference affinity and each group of fields

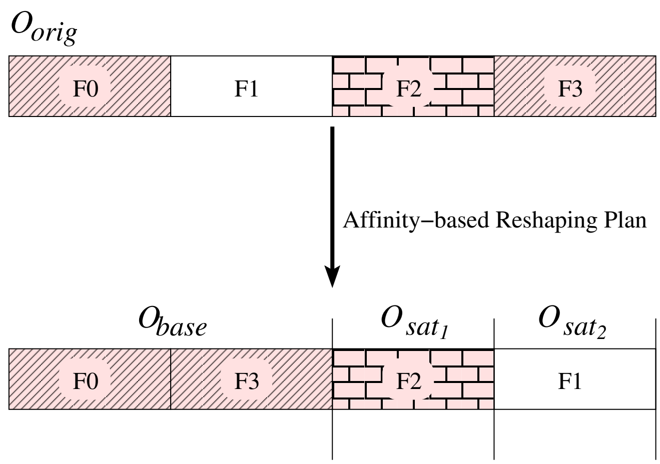


Fig. 3. Reshaping planning (affinity-based).

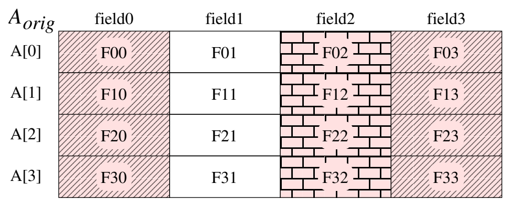
is split into an independent data structure. In Figure **??**, the original data structure is split into three new structures. The first structure contains fields F0 and F3. The second and third structures hold fields F2 and F1, respectively. Guided by the reshaping plan, a four-element array is split into three smaller arrays (Figure **??**).

ABP captures the runtime reference locality more closely than the other two planners discussed in this paper. Frequently, however, it is not easy to find clear-cut affinity relationships like the ones in Figure **??**. For example, a field A might have high reference affinity with two other fields B and C in different iterations, but B and C might never be accessed together. In such cases, it is difficult to find an optimal partition that respects competing affinities. The second drawback of affinity-based splitting is that hot fields without reference affinity may be placed into separate data structures and arrays. Placing hot fields in satellite arrays results in substantial address computation overhead.

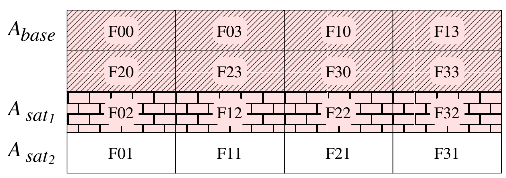
Frequency-Based Planner (FBP) or *hot-cold planner* uses runtime feedback information to partition a data structure into two. The first array contains hot fields and the second array contains cold elements. For instance, the first data structure in Figure **??** contains fields F0, F2 and F3. The second data structure contains field F1. A frequency-based planner only needs to calculate addresses of satellite fields that are infrequently accessed. Therefore, FBP does not require the execution of many additional instructions. However, it neither captures the reference affinity as well as ABP does nor reduces the memory footprint as aggressively as MSP, described below, does. Moreover, FBP might waste memory bandwidth and pollute the data cache. For instance, in the example of Figure **??**, when the program iterates through the fields F1, the fields F0 and F3 are uselessly fetched into cache.

Currently *Forma* uses 95% as a frequency-based planner threshold. That is, it retains the most frequently referenced fields that account for 95% of the accesses to the data structure in the base object. All other fields are treated as cold and are split to a satellite object.

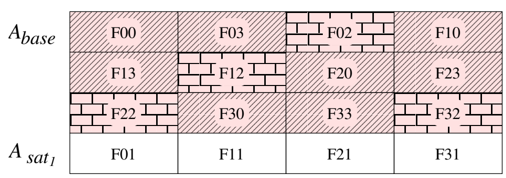
Maximal Splitting Planner (MSP) splits each field of a data structure into a



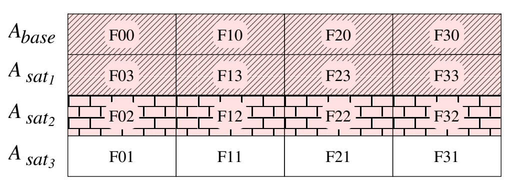
(a) Original Array



(b) Affinity-based Splitting.



(c) Frequency-based Splitting.



(d) Maximal Splitting.

Fig. 4. Different reshaping planning strategies.

separate new data structure, as shown in Figure **??**. After splitting, each data structure field is stored in an independent array. An obvious advantage of MSP is that it does not require profiling information. MSP ignores the reference affinity among the fields in the same data structure and seems to be too simple to be good for performance. Surprisingly, as shown in Section **??**, MSP achieves the best or close-to-best runtime performance among the different reshaping planners studied in this paper. This is because MSP has three important but subtle advantages.

First, MSP always achieves the smallest memory footprint for stride-1 iterations on the array. This is because each field has its own array and no irrelevant data is fetched into cache. In contrast, neither ABP nor FBP can guarantee that the partition respects access affinities for all traversals of the arrays. Therefore, even when *Forma* attempts to take into account affinity or frequency information, there may be traversals that fetch irrelevant fields into the cache. In modern processors, where the latency to bring data from memory is high, the smaller memory footprint generated by MSP can be a decisive advantage. For instance, it can significantly reduce misses in the lower levels of the memory hierarchy as well as in the translation look-aside buffer (TLB).

Second, MSP is especially suitable for processors that feature a hardware prefetching mechanism, such as the IBM POWER4™ processor [**?**]. A stream is a sequence of memory loads that access two or more contiguous data cache lines in either ascending or descending order. The processor monitors cache misses closely. Once misses on two consecutive cache lines are detected, a directed stream prefetching is triggered, and data from the memory area in the directed stream is fetched into higher levels of the memory hierarchy. The combination of hardware stream prefetching with high cache associativity allows independent streams that are accessed together to be simultaneously prefetched into different cache areas. This simultaneous prefetching tends to compensate for the loss of field affinity in MSP. Because the stride on each split array is smaller than those in arrays organized according to affinity, stream prefetching should work more efficiently. Therefore, there is no need to worry about the field affinity because the prefetching mechanism covers multiple streams consisting of fields with affinity. Fortunately, each processor supports eight independent streams, which seems to be sufficient for most applications. In the entire SPEC2000 benchmark suite, we haven't encountered any important array traversal that accesses more than eight fields. Moreover, even when there are more than eight streams in a loop, the XL compiler is able to distribute them into several smaller loops through loop fission [**?**].

Third, because maximal reshaping converts each field into a single object, the host object of a field contains only the field itself. Therefore, the address of the host object and the address of the field are the same and there is no need to compute the field offset. As a consequence, the address calculation for satellite fields is simpler when maximal reshaping is used.

MSP sacrifices field affinity to take advantage of field locality and reduce memory footprint. Loss of field affinity is compensated for by hardware stream prefetching and by higher associativity in modern cache systems.

A drawback of MSP is similar to the drawback of the affinity-based planner: all the fields, except the field in O_{base}, need to be accessed indirectly. Thus, if the field in O_{base} does not dominate the access frequency of the data structure, many additional instructions are executed to calculate the address of satellite objects.

2.4 Array Reshaping

The last phase of array reshaping transforms the program according to the reshaping plan. To apply array reshaping to an alias set, the allocation site and all related data accesses through pointers in the alias set should be transformed to reflect the change of the data view.

The design of *Forma* considered two transformation approaches: address-arithmetic-based splitting and pointer-based splitting. In address-arithmetic-based splitting, no extra fields are introduced during the transformation, and the address of a satellite field is calculated from its corresponding base object. Examples include the transformations shown in Figure ??, ??, and ??. In pointer-based splitting, extra field pointers are introduced in the base object to link each satellite object to its base object. Soon after the array is allocated, these extra field pointers are initialized to point to the corresponding satellite objects. Therefore, accesses to the satellite fields are transformed to accesses through the extra pointer dereferences.

Rule	Original	After address-arithmetic-based transformation
1	(O_{orig} *) p	(O_{base} *) p'
2	&(A[k])	&(A_{base}[k])
3	&(p -> basefield)	&(p' -> basefield')
4	&(A[k].basefield)	&(A_{base}[k].basefield')
5	&(p -> satfield)	index = (p'-A_{base})/LEN(O_{base}) &(A_{sat_i}[index].satfield')
6	&(A[k].satfield)	&(A_{sat_i}[k].satfield')
7	allocation site: A=new(N*E)	A_{base}=new(N*E) for i $\in$ [1, SatNum] A_{sat_i} = A_{base}+ offset$_i$*N

Table I. Address-arithmetic-based reshaping.

Rule	Original	After pointer-based transformation
1	(O_{orig} *) p	(O_{base} *) p'
2	&(A[k])	&(A_{base}[k])
3	&(p -> basefield)	&(p' -> basefield')
4	&(A[k].basefield)	&(A_{base}[k].basefield')
5	&(p -> satfield)	&(p' -> pointer$_i$ -> satfield')
6	&(A[k].satfield)	&(A_{base}[k].pointer$_i$ $\rightarrow$ satfield')
7	allocation site: A=new(N*E)	A_{base}=new(N*NewE) for i $\in$ [1, SatNum] for j $\in$ [0, N-1] A_{base}[j].pointer$_i$ = &(A_{sat_i}[j])

Table II. Pointer-based reshaping.

Tables ?? and ?? compares the differences between address-arithmetic-based reshaping and pointer-based reshaping. In the table, `p` is a pointer to an element in the array; `A` is the array's base address; `basefield` and `satfield` represent fields falling in O_{base} and O_{sat} in the plan, respectively. `SatNum` is the number of satellite arrays. `N` is the number of elements in the array. `E` is the size of each element in the original array and `NewE` is the size of the original element plus the sizes of the pointer fields introduced in a base object. The primed versions (such as `p'` and `satfield'`) represent their counterparts after reshaping. Tables ?? and ?? presents seven rules to transform references into reshaped objects and arrays.

—The 1st and 2nd rules say that, after reshaping, the role of the original object is taken by the base object. A pointer `p` that pointed to O_{orig} before reshaping is replaced by a pointer `p'` to O_{base} after reshaping. All the element-wise pointer manipulations are transformed to manipulate the base object. For example, `p++` means `p = p + sizeof(`O_{orig}`)` in the original program. It should be transformed to `p' = p' + sizeof(`O_{base}`)` after array reshaping.

—The transformation for the base fields (rules 3 and 4) is straightforward: their addresses are acquired by applying their new offsets in O_{base} to the pointer to O_{base}.

—The address calculation for satellite fields (rules 5 and 6) differs between the two approaches. In the pointer-based approach, the satellite fields are accessed via newly introduced pointer fields for the corresponding satellite object. In the address-arithmetic-based approach, the index of a satellite object equals $\frac{p'-A_{base}}{\mathtt{LEN}(O_{base})}$, where `LEN` is the length, in bytes, of O_{base}. This index is then used to access the corresponding element in the satellite array.

—The allocation site also has to be handled differently for the two approaches (rule 7). For address-arithmetic-based reshaping, one base pointer for each satellite array is introduced, and these base pointers are initialized after the array is allocated. In the pointer-based strategy, the program enumerates each element in the base array and initializes the pointer fields for the satellite objects.

Besides the type compatibility safety check, address-arithmetic-based splitting has another restriction to avoid unsafe transformations: single-instantiation. Single-instantiation restriction says that the entire alias set is instantiated by a single allocation site in the program and that the allocation site is executed no more than once at run time. This extra restriction makes sure that the base address of the array is a constant at run time.

Both address-arithmetic-based and pointer-based reshaping have their advantages and disadvantages. Pointer-based reshaping requires fewer address calculations and does not have the single-instantiation restriction. But it has two serious drawbacks. First, it requires extra pointer fields in each base object. When the reshaping plan splits O_{orig} into several new data structures, many extra fields are required. This additional data may offset the array reshaping effort. If the plan is frequency-based splitting, only one extra pointer field is needed. Second, each access to satellite fields requires one extra pointer dereference, which is often very expensive in today's register-centered processors.

In contrast, address-arithmetic-based reshaping only requires extra address calculations when the satellite fields are accessed via an element-wise pointer (rule 5). Therefore, if individual fields are accessed often, many additional address calculations would occur. However, this problem can be mitigated by traditional optimizations such as constant propagation, common subexpression elimination, and promotion of loop invariant expressions. For example, the address of $A_{sat_i}[\frac{p'-A_{base}}{\mathtt{LEN}(O_{base})}]$

is computed by the expression $A_{sat_i} + \frac{p' - A_{base}}{\texttt{LEN}(O_{base})} * \texttt{LEN}(O_{sat_i})$. At compile time, once reshaping is completed, $K = \frac{\texttt{LEN}(O_{sat_i})}{\texttt{LEN}(O_{base})}$ is constant. Consider the following extreme case that highlights the optimization potential for address-arithmetic-based reshaping: under maximal reshaping it is likely that $\texttt{LEN}(O_{sat_i})$ equals $\texttt{LEN}(O_{base})$ because both the O_{base} and O_{sat_i} contain a single field, and thus $K = 1$. In this case, `&(p -> satfield)`, that equals `p + OFFSET(satfield)`, is transformed to $A_{sat_i} + (p' - A_{base}) = p' + (A_{sat_i} - A_{base})$. The expression $(A_{sat_i} - A_{base})$ only needs to be computed once at the allocation site. In this extreme, but not infrequent, case the same number of operations are needed before and after reshaping!

Additionally, in address-arithmetic-based reshaping, the array size is exactly the same as the one before the transformation. Maintaining the same memory requirement makes the transformation safer than the pointer-based reshaping. The extra memory necessary for pointer fields may cause the program to fail when there is not enough free memory.

Although the address-arithmetic-based reshaping strategy requires single instantiation, this restriction is not a serious problem for array-centered applications. It may become a problem in applications that use linked data structures (LDS). Section **??** discusses analysis and transformations that will be required to deal with the multiple-instantiation problem in applications that use LDS extensively.

With all these factors taken into account, *Forma* favors the use of the address-arithmetic-based reshaping strategy.

3. EXPERIMENTAL STUDY

This section presents a performance study of array reshaping. The results of this investigation may be summarized as follows:

—Data reshaping improves data-intensive programs dramatically. This study found impressive performance gains — up to 2.1 times speedup — on three benchmarks. Maximum reshaping achieves best or close-to-best performance improvement.

—Data reshaping degrades the `perimeter` benchmark. A micro-architectural performance study reveals that in this benchmark additional instructions required for address calculation offset the small improvement on cache efficiency achieved by array reshaping.

—Maximum reshaping is best suited for programs with stride-1 iterations in architectures with hardware stream prefetching such as the IBM POWER™ family.

3.1 Experimental Platform

Forma is implemented in the IBM XL C/C++ V7.0 compiler. Thanks to the modular structure of *Forma*, it is easy to switch on different reshaping strategies and examine their effects. Table **??** shows the characteristics of the machines used in this performance study.

Three of these machines use processors from the IBM POWER family. The compiler is a development version of the IBM XL C/C++ that includes *Forma*. In order to investigate the portability of the results obtained with *Forma* in the XL compilers, the benchmarks were modified, by hand, and run in an Intel® Itanium-II machine. This hand modification consisted of inspecting the reshaping plans created by *Forma* and mimicking them in the benchmark's source codes. These benchmarks were then compiled with the Open Research Compiler 2.1 at the O3 optimization level with inter-procedural optimization. Although not applicable to a large number of benchmarks, this effort produced data that should convince developers of other compilers to consider when implementing array reshaping.

CPU	GHz	L1D, L2D, L3D, Mem	OS	Page Size
G5	2.0	32K, 512K, 0 , 1G	Darwin 7.5	4KB
POWER4	1.1	32K, 1.44M†, 32M †‡ , 32G	AIX® 5.2	4KB
POWER5™	1.65	32K, 1.92M†, 36M†‡ , 16G	AIX 5.3	4KB
Itanium-II	1.3	16K, 256K †, 1.5M† , 1G	Linux® 2.4.18	16KB

Table III. Characteristics of the experimental platforms, memory and page sizes given in bytes (†: DCache+ICache, ‡: off-die).

There is a limited number of benchmarks that are affected by array reshaping in standard benchmark suites. This study uses two benchmarks from the SPEC2K suite: `art` and `mcf`. Standard training data is used for profiling and the standard reference data for final runtime benchmarking. The memory footprints in the final run are 4.7MB for `art` and 190 MB for `mcf`. The other two benchmarks in this study, `tsp` and `perimeter`, are from the array version of OLDEN 1.3 benchmark suite. The profiling inputs used are 10^5 for `tsp` and 10 for `perimeter`. The inputs for the final runtime measurement are 4×10^6 for `tsp` and 11 for `perimeter`. The memory footprints for these final runs are 225MB and 256 MB, respectively.

Forma implements three partition planners and two reshaping methods. Table **??** lists the versions of the compiler that are included in this performance study along with the label identifying each version in all the graphs. Pointer-based reshaping requires many extra pointer fields in O_{base} when O_{orig} is broken into many satellite objects. These additional pointers seriously offset the benefit of reshaping. Therefore, pointer-based reshaping is not included in the performance study of affinity-based splitting and maximum splitting.

3.2 Run Time Improvement

Figures **??**-**??** presents the runtime variations among the array reshaping versions implemented in *Forma* in four hardware platforms. For each machine, the lower

Label	**Partition Planner**	**Reshaping Method**
baseline	—	—
affinity	affinity-based planner	address-arithmetic-based reshaping
freq-a	frequency-based planner	address-arithmetic-based reshaping
freq-p	frequency-based planner	pointer-based reshaping
max	maximal planner	address-arithmetic-based reshaping

Table IV. Compiler versions in the performance study.

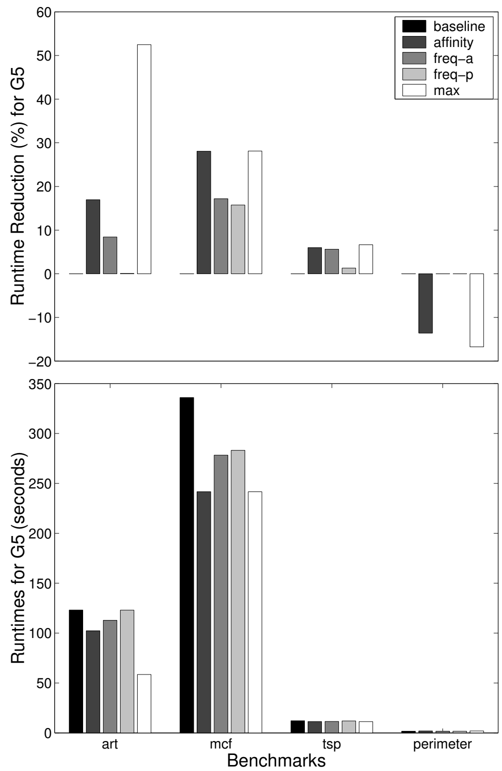


Fig. 5. Run times in G5.

graph presents the actual run times, and the upper graph is the normalized runtime percent variation. The baseline compiler is a development version of the industry-strong XL optimizing compiler running at optimization level O5 without any data reshaping optimization. For the Itanium, the baseline is the Open Research Compiler (ORC) at O3 optimization level with inter-procedural optimizations enabled. Array reshaping results in impressive performance improvement for `art`, `mcf`, and `tsp`.

In the G5 system, the maximal split version of `art` runs more than twice as fast as the baseline. In comparison, affinity-based reshaping improves `art` by about 17% and freq-a improves it by about 8.4%. The improvement by freq-p is negligible mainly because `art` has only one cold field, and the introduced pointer field completely nullifies the benefit of splitting this cold field. For `mcf`, affinity-based reshaping and maximum reshaping result in a performance improvement of about 28%. Both versions of frequency-based reshaping improve `mcf` by about 16%. Maximum reshaping and affinity-based reshaping improve `tsp` by 6.7% and 6%, respectively. For frequency-based reshaping, the improvements for `tsp` range from 5.6% for freq-a to 1.3% for freq-p. The only benchmark where data reshaping results in performance degradation is `perimeter`.[4] All the iterative behavior of `perimeter`

[4]In `perimeter`, the major data structure of interest is `quad_struct`. There is no infrequently

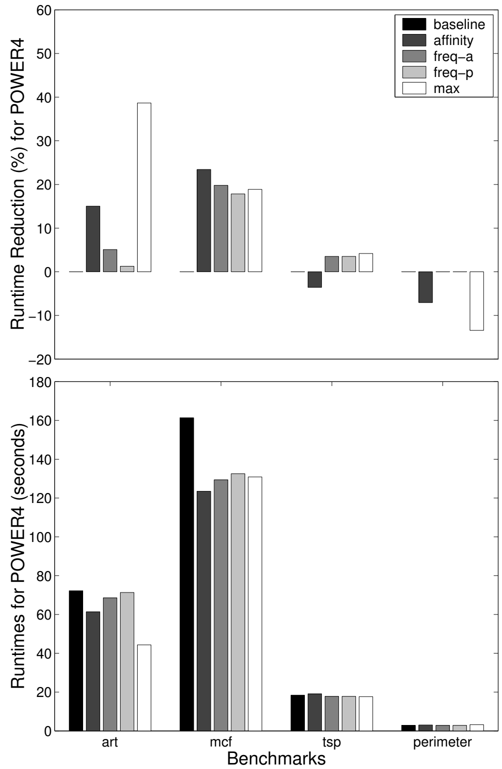


Fig. 6. Run times in POWER4.

results from recursive function calls as it does not contain any loops. Many common optimizations that reduce the cost of array reshaping, such as inlining and loop-invariant expression promotion, are difficult to apply to recursive code. Inlining is important because it enables further local optimizations. It is also difficult to lift common subexpressions from iterative code when the iterations are the product of recursion. A micro-benchmarking study, presented in Section ??, found that reshaping in `perimeter` significantly increased the number of instructions executed and had little effect on cache efficiency. Therefore, it is not surprising that data reshaping degrades the performance of `perimeter`. Future work will improve the reshaping heuristics to make them more conservative for recursive code.

The performance of array reshaping on POWER4 is similar to that on G5. The superior memory hierarchy in POWER4 reduces the impact of the poor memory reference in the baseline on the run time of the benchmarks. Therefore, though still impressive, the performance improvement is less significant than that on G5. The effect of a richer memory hierarchy on the baseline is even more pronounced on POWER5. As shown in Figure ??, freq-a performs better than affinity-based

accessed fields in `quad_struct`. Therefore the two versions of frequency-based reshaping choose not to do any reshaping. While this information could be added to the affinity-based reshaper heuristic, it may be desirable to keep the maximal reshaper independent of feedback information.

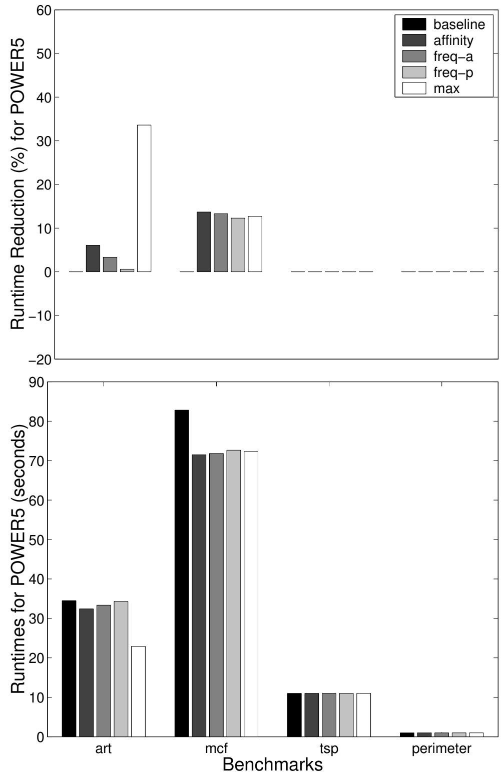


Fig. 7. Run times in POWER5.

reshaping on `tsp`. This result is the opposite of the findings in Zhong's work [?], where FBP almost always degrades performance for POWER4. Zhong *et al.* split fields that account for 50% of the total accesses into the base object. This threshold should be higher. *Forma* uses 95%. Compared with ABP and MSP, the only advantage of FBP is that it incurs negligible extra address calculation. A threshold of 50% in FBP generates many extra address calculations and thus eliminates its only advantage. Therefore, it is not surprising that the frequency-based reshaping is much worse than the affinity-based reshaping in [?].

The Itanium-II's architecture is very different from the architecture of the processors in the POWER family. The performance results reflect these differences. The performance improvement for `art` — 20% for maximal reshaping — is less impressive than that in the POWER family processors. This is because `art` often iterates on an array with stride 1, which best suits the hardware stream prefetching in the POWER processors. The most significant improvement in Itanium-II is a 45% reduction in run time for the maximal reshaping of `mcf`. Array reshaping in `tsp` results in more significant performance improvement in Itanium-II, from 12.4% to 22.6%, than in the POWER processors. The degradation of `perimeter` in Itanium-II indicates that the ORC 2.1 optimizations are also limited by the problems caused by recursion.

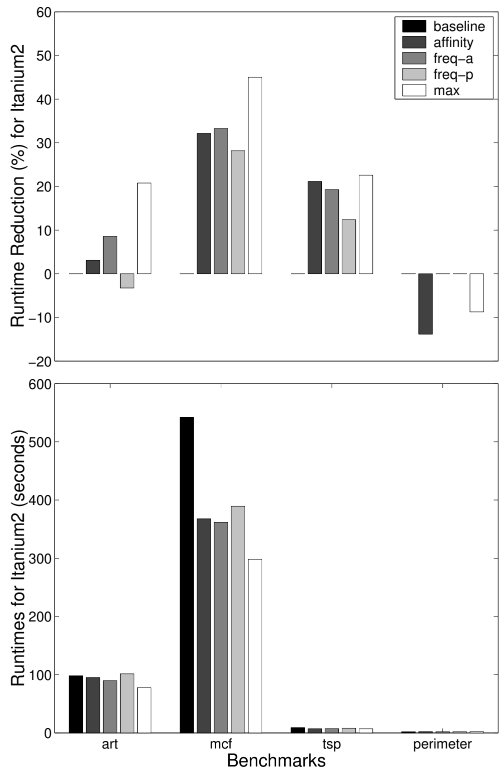


Fig. 8. Run times in Itanium II.

In summary, maximum reshaping performs best, or close-to-best, in all the studied strategies.

3.3 Micro-architecture Performance Study

This section presents measurements obtained with the `pfmon` performance monitoring tool in the Itanium-II workstation. These measurements further the understanding of the performance impact of data reshaping. This micro-architecture performance study examined the number of retired instructions, the miss rates at different cache levels, the TLB miss rates, and so on. This section presents only the measurements that showed a correlation with array reshaping.

The number of instructions retired by each version of array reshaping in the Itanium-II workstation, shown as a percentage variation over the baseline in Figure **??**, provides important insights on the effects of array reshaping at the micro-architectural level. Frequency-based reshaping has negligible instruction increases and results in fewer instructions retired in `tsp`. These measurements indicate that the profiling input data is indeed representative and allows the compiler to precisely identify cold fields. Affinity-based reshaping and maximum reshaping may significantly increase the number of retired instructions. The most significant results of this metric are the significant increase in the number of instructions executed for `perimeter` and the difference in the number of instructions executed for affinity-

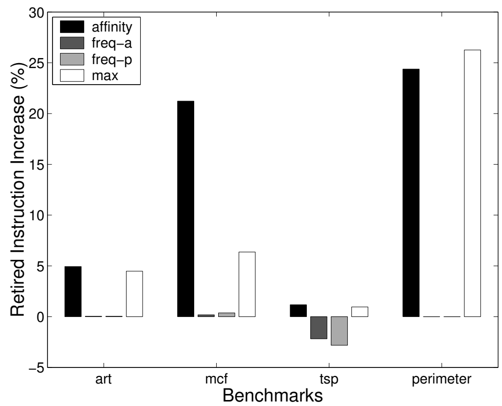


Fig. 9. Retired instructions on Itanium-II.

based and maximal reshaping. A comparison of this data with the execution times in Figure **??** reveals a positive correlation with the execution time of the benchmarks.

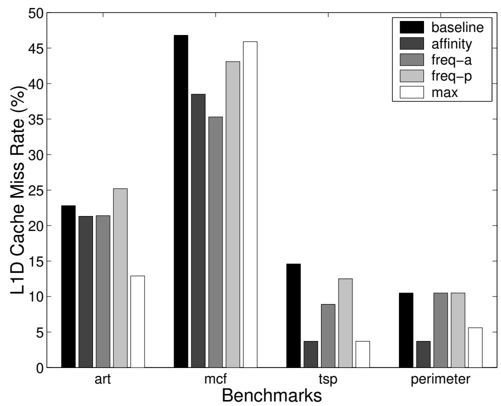
Figure **??** shows the first-level data cache miss rates. Maximal reshaping produces significantly fewer misses in `art`: the level 1 data cache (L1D) miss rate is reduced by 12.8% compared with baseline. For `mcf`, frequency-based and affinity-based reshaping reduce the L1D cache miss rate by 21.6% and 15.6%. The effect of maximal reshaping on level 1 data cache in `mcf` is quite small. For `tsp` and `perimeter`, maximal reshaping and affinity-based reshaping achieve similar improvement over baseline.

Figure **??** shows the second-level data cache miss rates. Different reshaping strategies make little difference on the L2D miss rate for `tsp` and `perimeter`. For `art` and `mcf`, the situation is similar to that in the L1D cache miss rate: maximal reshaping performs best on `art` and frequency-based reshaping is most effective on `mcf`. But the difference of effectiveness on `mcf` is not as significant as in L1D.

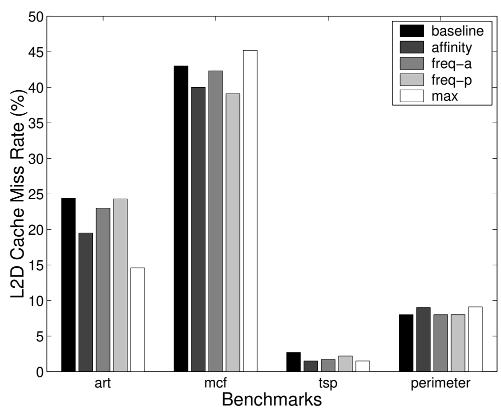
The L3D cache miss rates is shown in Figure **??**. There is no significant effect on the L3D miss rate of different reshaping strategies for `art`, `tsp` and `perimeter`. The important data in this graph is the significant reduction, 56.3%, in the L3D miss rate of `mcf` produced by maximal reshaping. With more than 40% access misses on both first- and second level-data cache, at least 16% of `mcf`'s data accesses turn to the third-level data cache. Therefore, the improvement on the L3D cache miss rate avoids a substantial number of memory references (about 9% of the total memory accesses). This improvement is important because the cost of physical memory accesses in this Itanium-II system is about 103.1 nanoseconds, which amounts to about 143 cycles[5].

TLB efficiency is another important performance factor for benchmarks with large memory footprints. The Itanium-II has two levels of data TLBs (DTLB). To translate a virtual address into a physical address, the processor first looks up the first-level DTLB (L1DTLB). If L1DTLB does not hold the mapping, a four-cycle

[5]This estimate for the physical memory access latency was obtained with `Lmbench 3.0`.



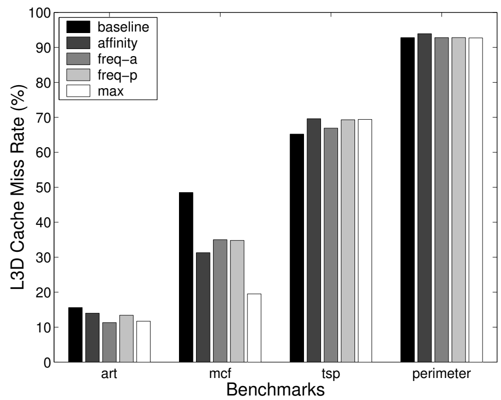
(a) L1D Cache Miss Rate on Itanium-II.



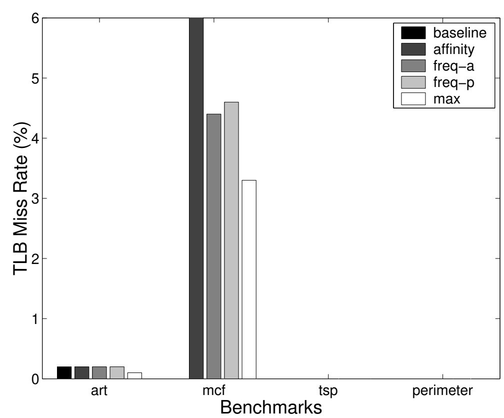
(b) L2D Cache Miss Rate.

Fig. 10. Data cache (levels 1 and 2) efficiency.

latency second-level DTLB (L2DTLB) access is required. If L2DTLB also fails, a TLB miss occurs. Itanium-II has a hardware virtual hash page table (VHPT) walker that reduces the overhead of TLB misses [**?**]. If the translation is not found in the VHPT, the execution traps to the operating system and a software handler is invoked to handle the TLB miss. Software handlers are extremely expensive because they execute hundreds of instructions. Of the four benchmarks studied, `mcf` is the only one that has a relatively high TLB miss rate. However, only about 0.5% `mcf`'s TLB misses result in software handler traps. All the other TLB misses are handled successfully by hardware VHPT walkers. The Itanium-II workstation used



(a) L3D Cache Miss Rate on Itanium-II.



(b) TLB Miss Rate on Itanium-II.

Fig. 11. Data cache (level 3) and TLB efficiency.

in this study has an average TLB miss latency of about 50 cycles[6]. Figure **??** shows the variations in TLB misses for all benchmarks and reshapers studied. Maximal reshaping reduces the TLB miss rate of `mcf` by 2.3%. This is because maximal reshaping always has the smallest memory footprint. The reduced TLB miss rate combined with the reduced DL3 miss rate (see Figure **??**) explain the impressive runtime improvement of maximal reshaping for `mcf` in the Itanium-II workstation (see Figure **??**).

[6]We used a program at http://www.gelato.unsw.edu.au/IA64wiki/PageFaultTiming to measure TLB miss latency.

4. RELATED WORK

Because data cache efficiency is a major performance bottleneck in modern computer systems, extensive research effort has been dedicated to improving data cache utilization. Extant research falls into three categories: data layout optimization, data prefetching, and loop restructuring. This section presents a sample of relevant work in this area.

4.1 Data Layout Optimization

The goal of data layout optimizations is to reduce data cache misses by improving data locality or by reducing cache conflicts.

4.1.1 *Structure Splitting or Reshaping.* Extensive research effort has been dedicated to the study of field placement and data splitting. Using the accumulated frequencies of the member fields, Franz *et al.* split an aggregate data type into a hot structure and a cold one [**?**]. Zhong *et al.* presents K-distance analysis to group fields in a structure according to their access affinity [**?**]. Both Franz and Zhong orient their techniques for type-safe programming languages. However, their performance studies use error-prone human-inspected C applications. Applying a type-safe oriented optimization to a type-unsafe language without proper alias analysis is dangerous in production compilers.

Rabbah *et al.* split a structure completely and group the respective fields of various data objects together [**?**; **?**], which is essentially the maximal splitting in this work. In their work, objects may be organized in an array, or may be linked through pointers in the original program. Their research has two shortcomings. First, they use imprecise field-insensitive alias analysis. With this conservative alias information, either important optimization opportunities will be missed or runtime checks must be inserted into the executable, potentially offsetting the data reshaping benefit. Moreover, their analysis does not include safety analysis.

4.1.2 *Array Padding and Array Permutation.* Inter-array padding adjusts array-based addresses by inserting memory space between arrays while intra-array padding modifies array dimensions by inserting spaces between array elements [**?**; **?**]. The motivation for array padding is to change the array layout so that array elements that are accessed at the same time are not mapped into the same cache address. Array padding is very useful for applications such as dense numerical linear algebra, finite-difference and partial differential equation solvers, and image processing. Array padding and array reshaping work with different data granularities. While array padding treats objects (such as *structs* and *classes*) as atomic, array reshaping works at the field level. Array reshaping is only good for arrays of large aggregate data elements. Moreover, the two techniques have different benefit models: array padding reduces cache conflict misses between frequently referenced data objects while array reshaping tries to avoid bringing useless data into cache.

Strip-mining and array permutation are used to reorganize data in multi-processor systems to make each individual processor's data share contiguous [**?**].

4.2 Loop Restructuring

Loop restructuring has been used to improve cache efficiency for a long time. Loop fusion might improve cache efficiency if both fused loops have access to the same data elements [?; ?; ?; ?; ?]. Loop fission, also called loop distribution, splits a loop into two or more smaller loops, each of which accesses independent arrays [?]. Loop interchange, also known as loop permutation, reorders the iterations over a multi-dimensional array so that the access pattern is more amenable to data layout [?; ?]. Loop tiling, also referred to as loop blocking, improves cache efficiency by dividing the iteration space of a loop into tiles that have better spatial and temporal locality [?; ?; ?; ?]. Loop tiling can only be applied to perfectly nested loops. Some imperfectly nested loops can be converted to perfectly nested ones so that tiling can be applied. Kodukula *et al.* proposes a "data-centric" approach, called data shackling or data blocking, to localize data accesses [?]. Intuitively, the compiler divides an array into a sequence of smaller blocks, as in loop tiling, and schedules, or *shackles*, the statements that operate on each block close together. At run time, once the data block is fetched into memory hierarchy, the shackled statements are all executed.

All these loop restructuring techniques are compile-time optimizations. These techniques require that access patterns be known to the compiler. However, in some applications the access patterns are hard to predict at compile time. To deal with these situations, researchers have proposed runtime data layout or control flow transformations [?; ?]. However, these approaches have two drawbacks. First, they often ignore practical problems, such as alias analysis for safety, that are indispensable in a production compiler. Second, the optimization targets are often very specific, and reducing the overhead for general runtime transformation is still an open question.

4.3 Data Prefetching

Data Prefetching can be seen as an orthogonal optimization to data reshaping. Even though they share the goal of alleviating the memory bottleneck problem by reducing memory latency, data prefetching and data reshaping approach the problem from different angles. Data prefetching is a technique to *tolerate* memory latency by loading data into the cache when the data is expected to be used soon. Data prefetching does not reduce cache misses; rather it reduces the cache miss penalty. Data prefetching tries to handle a cache miss earlier by overlapping the data fetching with other computations so that the data is already in cache when it is needed. Comparatively, data reshaping reduces memory latency by improving data locality and *improving* cache hit rates.

Data prefetching has been studied extensively. A good survey can be found in [?]. Data prefetching can be either hardware based, software based [?; ?; ?] or a joint effort of hardware and compiler support [?; ?].

The drawback of data prefetching is that it may increase substantially the number of memory accesses and the demand for memory bandwidth. This problem becomes dominant in systems where interconnections to a memory subsystem are shared by several processors. From this perspective, data reshaping is superior to data prefetching because it attacks the problem at its cause, by reducing misses, instead of at its observed effect, *i.e.* by tolerating misses.

Badawy *et al.* found that software data prefetching outperforms loop restructuring when there is enough memory bandwidth available [?]. In the same work, they also found that naively integrating software prefetching with loop restructuring does not yield additional performance improvements.

Sometimes the effects of data prefetching and reshaping might overlap. With careful data layout or reshaping, the effect of data prefetching might become smaller because the reshaped data has better locality and is likely to be in cache already when it is referenced, thus eliminating the need for prefetching. If the cache miss pattern can be predicted well by the data prefetching mechanism, data reshaping is not necessary unless memory bandwidth becomes a problem. However, data prefetching and reshaping are not necessarily mutually exclusive. Data reshaping might also facilitate data prefetching. The cooperation between maximal splitting and hardware stream prefetching in PowerPC® processors is a good example.

5. CONCLUSION AND FUTURE WORK

Forma is a practical array reshaping framework that guarantees safe automatic array reorganization. The experimental evaluation of *Forma* studied the effects of design decisions on two dimensions: the reshaping planner and the reshaping method.

Forma has limitations. Although it catches important cases in standard benchmarks and produces impressive performance improvements, the single instantiation rule for array reshaping is restrictive. Because of this restriction, currently *Forma* only handles dynamically allocated arrays. However, many programs operate on linked data structures that are typically not allocated monolithically into a dynamic array. The next step on the development of *Forma* is to handle these cases. The challenge to handle individually allocated objects is the difficulty to analyze all the dynamically allocated objects involved in a linked data structure at compile time. Therefore, *Forma* will need to insert a sophisticated memory pool management mechanism into the program and integrate this mechanism into the programs' memory management. Recently, there has been some work in this direction [**?**; **?**; **?**]. Adding this feature will increase the number of opportunities for reshaping covered by *Forma*.

A common shortcoming of existing automatic data layout optimization techniques is that they only capture very simple access patterns or use imprecise approximations. To accommodate more complex data access patterns, researchers in the area of cache-conscious algorithms have manually re-engineered applications [**?**; **?**; **?**]. Re-crafting an algorithm can produce impressive performance improvements but its high cost makes it prohibitive for a large number of applications. It will be interesting to compare the performance potential of existing automatic reshaping strategies with complex manual transformations. If these manual transformations are general and much more superior, we should investigate more sophisticated program analysis techniques that are necessary to automate this process. These techniques include phase recognition, access pattern (or shape) analysis, and so on. As the impact of memory accesses on performance grows, automatic data reorganization will be justified in spite of its complexity.

6. ACKNOWLEDGEMENTS

We thank the Toronto Portable Optimizer (TPO) teams for building the infrastructure on which we conducted this research. Special thanks to Christopher Barton and Roch Archambault for extensive and fruitful discussions. Thanks also go to Yutao Zhong, Chris Lattner *et al.* for discussions about OLDEN benchmarks. We also thank Stephane Eranian, Ian Wienand *et al.* for their help with Itanium-II benchmarking. This research was conducted during an internship at the IBM Toronto Laboratory and was partly supported through an IBM Faculty Award, as well as by the National Sciences and Engineering Council of Canada (NSERC) through a Collaborative Research and Development (CRD) grant.

7. TRADEMARKS

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both: AIX, IBM, POWER, PowerPC, POWER4, POWER5. UNIX is a registered trademark of The Open Group in the United States and other countries. Intel is a registered trademark of Intel Corporation in the United States, other countries, or both. Linux is a trademark of Linus Torvalds in the United States, other countries, or both. Other company, product, and service names may be trademarks or service marks of others.

REFERENCES

H. Al-Sukhni, I. Bratt, and D. A. Connors. Compiler-directed content-aware prefetching for dynamic data structures. In *12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 91–100, New Orleans, LA, 2003.

J. R. Allen and K. Kennedy. Automatic loop interchange. In *SIGPLAN Symposium on Compiler Construction*, pages 233–246, Montreal, QC, Canada, 1984.

A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In 2001 International Conference on Supercomputing (ICS'01) , pages 486–500, Sorrento, Italy, 2001.

C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, pages 13–24, Berlin, Germany, June 2002.

J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. Ccured in the real world. In *Programming Language Design and Implementation (PLDI)*, pages 232–244, San Diego, CA, June 2003.

C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Programming Language Design and Implementation (PLDI)*, pages 229–241, 1999.

M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Programming Language Design and Implementation (PLDI)*, pages 242–256, Orlando, FL, 1994.

M. Franz and T. Kistler. Splitting data objects to increase cache utilization. Tech. Report ICS-TR-98-34, Dept. of Information and Computer Science, Univ. of California, Irvine, Oct 1998.

M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, Portland, OR, 2000.

R. C. Holte, T. Mkadmi, R. M. Zimmer, and A. J. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1–2):321–361, 1996.

C. Hsu and U. Kremer. A stable and efficient loop tiling algorithm. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, Newark, DE, 2000.

http://www.redbooks.ibm.com/. *The Power4® Processor Introduction and Tuning Guide*. IBM Corp, International Technical Support Organization, 2001.

Intel. *Intel® Itanium® Architecture Software Developer's Manual*. Intel Corp, 2002.

K. Ishizaka, M. Obata, and H. Kasahara. Cache optimization for coarse grain task parallel processing using inter-array padding. In *Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 64–76, College Station, TX, Oct 2003.

ISO/IEC. *International Standard ISO/IEC 9899, Programming Languages - C. 1st Edition*. 1990.

S. P. Amarasinghe J. M. Anderson and M. S. Lam. Data and computation transformations for multiprocessors. In *Principles of Programming Languages (POPL)*, pages 166–178, Santa Barbara, CA, July 1995.

Magnus Karlsson, Fredrik Dahlgren, and Per Stenstrom. A prefetching technique for irregular accesses to linked data structures. In *6th International Symposium on High-Performance Computer Architecture*, pages 206–217, 2000.

K. Kennedy. Fast greedy weighted fusion. In *14th International Conference on Supercomputing*, pages 131–140, Santa Fe, NM, 2000.

I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Programming Language Design and Implementation (PLDI)*, pages 346–357, Las Vegas, NV, May 1997.

C. K. Luk. *Optimizing the Cache Performance of Non-Numeric Applications*. PhD thesis, Univ. of Toronto, Dept. of Computer Science, February 2000.

C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 222–233, Cambridge, MA, October 1996.

K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Principles of Programming Languages (POPL)*, pages 128–139, Portland, OR, Jan 2002.

R. Niewiadomski, J. N. Amaral, and R. Holte. Crafting data structures: A study of reference locality in refinement-based path finding. In *International Conference on High Performance Computing*, pages 438–448, Hyderabad, India, December 2003.

R. Niewiadomski, J. N. Amaral, and R. C. Holte. A performance study of data layout techniques for improving data locality in refinement-based pathfinding. *The ACM Journal of Experimental Algorithmics*, 2004. To appear.

S. Palem, R. Rabbah, P. Korkmaz V. J. Mooney, and K. Puttaswamy. Design space optimization of embedded memory systems via data remapping. In *2002 Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES'02-SCOPES'02)*, pages 28–37, Berlin, Germany, June 2000.

R. Rabbah and S. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions Embedded Computing System*, 2(2):186–218, 2003.

G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Programming Language Design and Implementation (PLDI)*, pages 38–49, 1998.

G. Rivera and C.-W. Tseng. A comparison of compiler tiling algorithms. In *8th International Conference on Compiler Construction*, pages 168–182, 1999.

Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. *ACM SIGPLAN Notices*, 33(11):115–126, 1998.

B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, pages 168–179, 2003.

S. Singhai and K. McKinley. Loop fusion for data locality and parallelism. In *Mid-Atlantic Student Workshop on Programming Languages and Systems*, New Paltz, NY, April 1996.

S. Singhai and K. S. McKinley. A parameterized loop fusion algorithm for improving parallelism and cache locality. *The Computer Journal*, 40(6):340–355, 1997.

B. Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *6th International Conference on Compiler Construction*, pages 136–150, 1996.

B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Programming Languages (POPL)*, pages 32–41, 1996.

A. Stoutchinin, J. N. Amaral, G. R. Gao, J. Dehnert, S. Jain, and A. Douillet. Speculative prefetching of induction pointers. In *International Conference on Compiler Construction 2001*, pages 289–303, 2001.

M. M. Strout, L. Carter, and J. Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Programming Language Design and Implementation (PLDI)*, pages 91–102, San Diego, CA, June 2003.

S. P. VanderWiel and D. J. Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

M. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, 1987.

M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

S. H. Yong, S. Horwitz, and T. Reps. Pointer analysis for programs with structures and casting. In *Programming Language Design and Implementation (PLDI)*, pages 91–103, Atlanta, GA, 1999.

Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Programming Language Design and Implementation (PLDI)*, pages 255–266, Washington, DC, June 2004.