University of Alberta

Library Release Form

Name of Author: Stephen Matthew Curial

Title of Thesis: Safe Automatic Data Splitting for Linked Data Structures

Degree: Master of Science

Year this Degree Granted: 2007

Stephen Matthew Curial

Date: _____

University of Alberta

SAFE AUTOMATIC DATA SPLITTING FOR LINKED DATA STRUCTURES

by

Stephen Matthew Curial

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

in

Department of Computing Science

Edmonton, Alberta
Fall 2007

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Safe Automatic Data Splitting for Linked Data Structures** submitted by Stephen Matthew Curial in partial fulfillment of the requirements for the degree of **Master of Science** in .

José Nelson Amaral (Supervisor)

Jonathan Schaeffer

Duncan Elliott (External)

Date: _____

Abstract

Data transformations, such as pool allocation, have shown promise as a method of reducing the number of cache misses for dynamically allocated data in non-numeric applications. To further improve data locality this thesis proposes a data reorganization technique that extends pool allocation by incorporating maximal structure splitting.

We develop a safe, fully automatic technique known as Memory-Pooling-Assisted Data Splitting (MPADS) that can improve spatial locality for pointer-based data structures allocated in the heap. To change the storage location of dynamically allocated data, a memory allocation library is created and the address calculation of the data accesses are updated. MPADS is implemented in the IBM XL production compiler suite.

MPADS is evaluated on the SPEC 2000, Olden and LLU benchmark suites. The pointer analysis used in MPADS proves to be too conservative for the complex SPEC benchmarks and many opportunities are abandoned, but identifies opportunities in the Olden and LLU benchmarks. MPADS outperformed pool allocation for every benchmark tested and for one benchmark, MPADS increased performance by over a factor of 2 from the baseline.

Acknowledgements

First and foremost I would like to thank my supervisor Dr. José Nelson Amaral for providing me with guidance and encouragement during my studies. Your mentoring allowed me to grow as a researcher and as a person. You taught me the skills necessary to be successful as a graduate student. Your openness allowed me to tackle a problem that I was genuinely interested in and you showed me how important it is to be passionate for what you are working on. Thank you for everything you have done for me and helped me with during my studies.

I would also like to thank my colleagues at IBM, Dr. Yaoqing Gao, Shimin Cui, Raúl Silvera, Roch G. Archambault, Dr. Peng Zhao and Christopher Barton for helping me develop the idea behind structure splitting and guiding me through the intricacies of the the IBM XL compiler. Everyone I worked with took time to answer my questions and help me with any problems that I encountered. I am especially grateful to Dr. Peng Zhao who always made sure that I was making progress and spent a considerable amount of time working with me to implement the structure splitting optimization. Kelly Lyons, Marin Litoiu and Roland Koo should also be acknowledged because they helped me get accustomed to working at IBM and made me feel welcome in the lab.

My committee not only reviewed my thesis but provided many insightful comments and possible extensions to the work. I am grateful for Dr. José Nelson Amaral, Dr. Jonathan Schaeffer and Dr. Duncan Elliott's attention to detail and dedication.

IBM partially funded my studies and allowed me to move to Toronto to work with their compiler development group at the lab. They also provided me access to the IBM XL compiler and assisted me with my implementation. Without their support I would have been unable to develop the structure splitting transformation in a production compiler.

This work was also funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) through a Collaborative Research and Development (CRD) Grant.

Finally, I need to thank my parents, who's never ending support allowed me to wholeheartedly pursue my studies. I am grateful to them for the love and encouragement that they provided. As a child, I remember my Mom telling me "There's no problem in life so great it can't be solved." These words have stuck with me and provided me with the motivation and determination that allowed me to succeed with my research.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

From the 1960s until 2002, hardware manufacturers have kept pace with Moore's law, roughly doubling the number of transistors, and correspondingly, the processor's performance every 18 months [47] but traditional techniques to increase performance are yielding diminishing returns [3, 33, 69].

Modern architectures are reaching the limits of thermal density, power efficiency, instruction level parallelism and frequency scaling. While techniques such as pipelining and frequency scaling have worked well in the past, they are insufficient to keep pace with Moore's law. One of the major causes of the diminishing returns is that processor speeds are increasing dramatically faster than memory speeds. This problem is known as the memory wall [69].

Modern architectures can take hundreds of cycles to fetch data from main memory because memory speeds have not been able to keep pace with processor speeds. Ghoting *et al.* demonstrate the effects of the memory wall by increasing the CPU clock frequency by a factor of 2.38 (from 1300MHz to 3100MHz) but only obtain a performance speedup of 1.6 [17]. The limits of performance scaling for this example are a direct result of the naive data layouts used by compilers. If a better data organization could be realized the performance scaling may continue for faster clock speeds.

This thesis develops a technique to reorganize dynamically-allocated heap data to reduce the effects of the memory wall. The transformation targets general purpose applications that use link-based structures, such as linked lists, trees and graphs.

The data reorganization is fully automatic and guaranteed to not affect the semantics of the program. It doesn't require profile or trace information and it has been implemented in the IBM XL production compiler. To transform the data, two different splitting techniques have been designed to improve data locality and cache performance. The techniques trade address computation overhead for data locality.

1.1 Contributions

The structure splitting presented in this thesis can be viewed as an extension of Lattner and Adve's pool allocation [29]. The identification of candidate structures that are safe to transform has been slightly modified but is based on Lattner and Adve's Data Structure Analysis (DSA) that identifies type-homogeneous structures. Lattner and Adve implement DSA and pool allocation in the LLVM compiler Infrastructure that is based on GCC [28].

When this research was started the only published structure splitting techniques were Zhong *et al.* [71] and Rabbah and Palem [51]. Neither of these techniques could safely and automatically split a pointer-based data structure and both required trace or profile information. Both of these structure-splitting techniques perform affinity-based splitting and this research aimed to develop a technique for maximal splitting. Rabbah and Palem's framework had another drawback. It adds padding to the data structures that pollutes the cache.

To address the shortcomings of these systems, Memory-Pooling-Assisted Data Splitting (MPADS) was developed. MPADS combines an analysis to identify structures that are safe to transform and does not require profile information or a memory trace. MPADS performs maximal splitting instead of affinity-based splitting and does not add padding to any structure.

In April of 2007 Jeon, Shin and Han published their structure-splitting work at the Compiler Construction conference [24]. The structure splitting framework that they created is similar to the non-uniform splitting described in Chapter 3.3.2. Their framework uses a static analysis to determine the field-access affinity and performs affinity-based splitting.

A more detailed comparison of these systems with MPADS is given in chapter 6.

1.2 Thesis Organization

Chapter 2 gives the background information necessary and explains pointer analysis. The structure splitting transformation is described in Chapter 3, including the identification of safe structures to transform in Chapter 3.1. The two types of splitting, uniform and non-uniform are described in Chapters 3.3.1 and 3.3.2. Chapter 4 motivates the creation of the splitting transformation with several micro-benchmarks that demonstrate that it works. Structure splitting is applied to standard benchmarks and the performance is evaluated in Chapter 5. In Chapter 6 the related work is presented before the future work is described in Chapter 7. Chapter 8 concludes the thesis.

Chapter 2

Background

2.1 Definitions

Affinity – The likelihood that two memory locations will be accessed close in time. Two memory locations are said to have good affinity if they are often referenced together.

Aliased – Two pointers are aliased if the objects that they may point to can overlap or occupy the same memory location.

Basic Block – A consecutive group of instructions where the control flow must start at the first instruction and execute every instruction in the group until the last. Only the first instruction in a basic block can be the target of a branch and only the last statement of a basic block may be a branch.

Cycles Per Instruction – Cycles Per Instruction (CPI) are calculated as the quotient of the number of cycles divided by the number of instructions completed.

Data Dependence – A data dependence occurs when one instruction uses, or defines, either the result of another instruction or one of the operands of another instruction.

Intermediate Representation – An intermediate representation (IR) is a representation of a program that is independent of the language that the program was written in and independent of the machine the program will be executed on. Typically the IR for a compiler is either stack-based [18], three-operand-based [49] or tree-based [48]. Static Single Assignment (SSA) is a increasingly common IR in compilers [13].

Interprocedural – Refers to looking at the entire program rather than just a single procedure or region.

Killed – In pointer and data flow analysis the definition of a variable is killed along a path if there is an assignment to that variable that is always performed. For example, consider the code in figure 2.1. On line 1 x is defined with the value 6 but that definition is killed by the assignment on line 3.

```
BASICBLOCK B()
1  x = 6;
2  y = 7;
3  x = y;
```

Figure 2.1: Example to illustrate killing a definition of a variable.

Point – Points in a basic block occur before the first statement, in-between consecutive statements and after the last statement.

Spatial Locality – Two pieces of data or memory locations are said to have good spatial locality if they are located near each other.

Temporal Locality – Two memory locations are said to have good temporal locality if they are both frequently accessed in a short time frame.

Working Set – The working set of a process is the set of pages that the process accesses during the last λ memory references.

2.2 Pointer Analysis

To analyze and transform programs written in languages with pointers or references, such as C, C++ and Java, often the targets of the pointers must be known. Computing a precise runtime relation between pointers and their target locations is infeasible and a conservative approximation is commonly used. Pointer analysis[1] provides a static abstraction of the possible targets for each pointer in the program [15, 55]. There is a trade-off between the precision of the analysis and the time and space required to perform the analysis. The precision of a given alias analysis is often classified in three main dimensions, namely flow-sensitivity, context-sensitivity and field-sensitivity [55].

A flow-insensitive pointer analysis algorithm does not consider the order in which statements are executed. The analysis computes a single points-to set for each variable and the points-to set must be accurate for the entire execution of the program. A flow-insensitive analysis cannot exploit the fact that a variable may be killed, also known as a strong update, because the points-to set must be valid at every point in the program.

A flow-sensitive analysis does consider the order that the statements are executed. The analysis computes a points-to set for each variable at every point in the program. Storing a points-to set for each pointer variable in the program at every statement is very expensive and can significantly increase the space requirements of the analysis. Often the intermediate representation in the compiler will rename all of the variables so that each variable is defined only once. Performing

[1] Pointer analysis is also known as points-to analysis, reference analysis or refers-to analysis.

a flow-insensitive analysis on an intermediate representation with variable renaming will result in close to the same precision as a flow-sensitive analysis [22, 37].

```
VOID MAIN()
1  int * a = malloc(...); //Allocation site A
2  int * b = malloc(...); //Allocation site B
3  int * c = malloc(...); //Allocation site C
4  int * d = malloc(...); //Allocation site D
5  int * e = malloc(...); //Allocation site E
6  a = b;
7  foo(&c, &d);
8  foo(&e, &c);

VOID FOO(int **p1, int **p2)
1  *p1 = *p2;
```

Figure 2.2: Example program to illustrate the different abstractions used for pointer analysis.

To clearly describe the differences in precision due to flow-sensitivity consider the example program in Figure 2.2. To illustrate the differences, it is sufficient to only consider the points-to set for a. The flow-sensitive points-to set for a is: empty before line 1;$\{A\}$ between line 1 and 6; and $\{B\}$ after line 6. The assignment on line 1 adds A to the points-to set of a. The assignment on line 6 adds everything from the points-to set of b to the points-to set of a. In a flow-sensitive analysis, the assignment on line 6 also kills the previous points-to set of a, therefore the flow-sensitive points-to set for a after line 6 is executed contains only B. The flow-insensitive points-to set of a is $\{A, B\}$ for all points in the program.

Each time a procedure is called from a different calling context, it may take different parameters or return different objects. A context-sensitive analysis will analyze each procedure separately for every calling context. The context-insensitive analysis uses a single points-to set to model the parameters and return value of the callee. [2]

Once again, consider a flow-insensitive analysis of the example program in Figure 2.2. The function *foo* takes two parameters, $p1$ and $p2$. An assignment is performed resulting in $p1$ pointing-to $p2$. A context-sensitive analysis would use a separate points-to set for the parameters on lines 7 and 8. Thus for a context-sensitive, and inter-procedural, analysis line 7 adds D to the points-to set of c and line 8 adds the points-to set of c, $\{C, D\}$, to the points-to set of e. Therefore a context-sensitive analysis could determine that the points-to set of c is $\{C, D\}$ and the points-to set of e is $\{C, D, E\}$ because it differentiates between calling contexts. However a context-insensitive analysis uses a single points-to set for every calling context, thus the points to set for $p1$ and $p2$ are $\{C, E\}$ and $\{C, D, E\}$, respectively. Accordingly, the context-insensitive points-to set for c and e are calculated as $\{C, D, E\}$ and $\{C, D, E\}$, respectively.

[2]The callee is compiler-speak for the function that is called at a given call site. The function where the call site is located is the caller.

Table 2.1 shows the points-to sets for c and e in Figure 2.2. The results are given varying the context- and flow-sensitivity. For the flow-sensitive analysis the points-to sets correspond to the points between lines 6-7, 7-8 and after line 8. The precise context- and flow-sensitive analyzes have the smallest points-to sets for each pointer. Conversely the context- and flow-insensitive analyzes have the least precision and the largest points-to sets.

		flow	
		insensitive	sensitive
context	insensitive	$pt(c) = \{C, D, E\}$ $pt(e) = \{C, D, E\}$	$pt_{6-7}(c) = \{C\}$ $pt_{6-7}(e) = \{E\}$ $pt_{7-8}(c) = \{C, D\}$ $pt_{7-8}(e) = \{E\}$ $pt_{8}(c) = \{C, D\}$ $pt_{8}(e) = \{C, D\}$
	sensitive	$pt(c) = \{C, D\}$ $pt(e) = \{C, D, E\}$	$pt_{6-7}(c) = \{C\}$ $pt_{6-7}(e) = \{E\}$ $pt_{7-8}(c) = \{D\}$ $pt_{7-8}(e) = \{E\}$ $pt_{8}(c) = \{D\}$ $pt_{8}(e) = \{D\}$

Table 2.1: The points-to set for c and e from Figure 2.2. The flow-insensitive results are valid for the entire program while the flow-sensitive results given for the points between lines 6-7, 7-8 and after line 8.

If a pointer analysis differentiates between the accesses to each field of a structure then it is a field-sensitive analysis. A field-insensitive analysis will treat a structure as a single object and each access to a field is a reference to the entire object. For the example program in Figure 2.3 the points-to sets of $p->f1$ and $p->f2$ are the same set for a field-insensitive analysis, namely $\{P, Q\}$. The field-sensitive points-to set for $p->f1$ is $\{P\}$ and $p->f2$ is $\{Q\}$.

```
MAIN()
1  struct s1{
2     struct s1  *f1;
3     struct s1  *f2;
4  };
5  struct s1 *p = malloc(...); //Allocation site P
6  struct s1 *q = malloc(...); //Allocation site Q
7  p->f1 = p;
8  p->f2 = q;
```

Figure 2.3: Example program to illustrate field-sensitivity in pointer analysis.

Steensgaard's alias analysis is widely used because the runtime complexity is nearly linear with the size of the code, the storage requirements scale well to large programs and often it provides the required precision [23, 61]. The analysis is unification-based, meaning that when there is a pointer assignment, the right-hand side pointer is added to the alias set of the left-hand side and both of the

alias sets are unified. The unification allows the algorithm to avoid having to propagate changes in the points-to set and the algorithm can reach a solution without the numerous iterations required to reach a fixed-point solution. Steensgaard's alias analysis is flow-insensitive and context-insensitive. It can be implemented to be field-sensitive.

For example, consider the program given in Figure 2.4. A diagram of the points-to analysis is given in Figure 2.5. Figure 2.5(a) shows the points to sets in the algorithm after processing line 5 in Figure 2.4. The alias sets of b and c are unified in Figure 2.5(b) as a result of assignments on lines 5 and 6. Figure 2.5(c), (d), (e) and (f) correspond to processing lines 7, 8, 9 and 10 in Figure 2.4, respectively. The final points-to set for a is $\{b, c, d, e, f, g\}$.

VOID MAIN()
1 $int * a;$
2 $int * b = malloc(...);\ int * c = malloc(...);$
3 $int * d = malloc(...);\ int * e = malloc(...);$
4 $int * f = malloc(...);\ int * g = malloc(...);$
5 $a = b;$
6 $a = c;$
7 $d = e;$
8 $c = e;$
9 $e = f;$
10 $g = f;$

Figure 2.4: Example program to illustrate Steensgaard's pointer analysis.

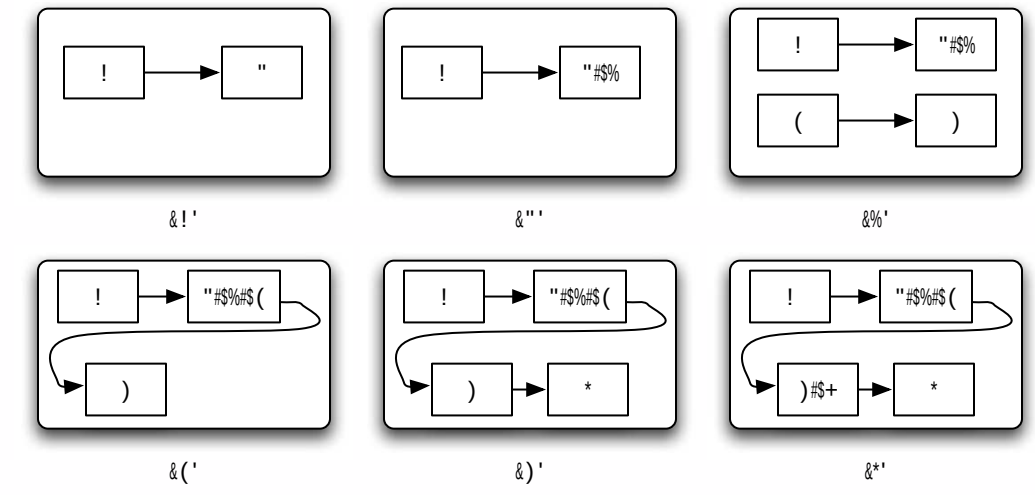


Figure 2.5: Steensgaard's pointer analysis for the program in Figure 2.4.

Many researchers are developing techniques to make precise pointer analysis scale well and only recently is pointer analysis with greater precision becoming a practical option. Binary Decision Diagrams (BDDs) have been proposed as a space-efficient method to represent relations in pointer

analysis [4, 38, 68, 73, 72]. Alternatives to BDDs such as Zero Suppressed BDDs (ZBDDs) and Don't-Care BDDs (XBDDs) have also been proposed as another space efficient representation [35]. Hardekopf and Lin eliminate cycles from inclusion-based pointer analysis and reduce the running time of the analysis so it can be performed on large programs in a reasonable amount of time [21]. Lattner, Lenhart and Adve show that a context- and field-sensitive analysis can scale to programs with hundreds of thousands lines of code [31].

2.3 Compiler Infrastructure

The IBM XL Compiler suite is a widely used production compiler that supports C, C++, Unified Parallel C (UPC) and Fortran. The XL Compiler suite is a highly developed compiler with an aggressive program transformation and optimization framework. The compiler is comprised of three main components: the front-end, the middle-end and the back-end. Information is exchanged between components using an intermediate representation known as W-code.

The front-end of the compiler is also known as the parser. The XL compiler suite has a separate parser for each of the languages supported. The front-end parses the source code and converts it to the language-independent W-code intermediate representation. The W-code is then either sent to the middle-end or the back-end depending on the desired level of program optimization.

If the user is willing to trade compile time for a faster executable, then the middle-end is invoked. The middle-end in the XL compiler is a link-time optimizer and is also known as the whole-program optimizer or the Toronto Portable Optimizer (TPO). The TPO performs many important optimizations such as inlining, loop optimizations and inter-procedural analysis.

The back-end, also known as the code generator or TOBEY (Toronto Optimizing Back End with Yorktown), takes the intermediate representation from the front-end or middle-end and generates machine code for the target platform. TOBEY performs optimizations such as instruction scheduling, constant folding and register allocation.

Chapter 3

Memory-Pooling-Assisted Data Splitting

Splitting data structures is a data reorganization technique that can significantly increase the spatial locality of data and reduce the runtime of programs that use link-based data structures [10, 14, 26, 63].

Memory-Pooling-Assisted Data Splitting (MPADS) is a framework designed to safely and automatically split pointer-based data structures without adding padding. MPADS uses a pointer analysis that enables the compiler to guarantee that it can safely split a given structure even when the program is written in a weakly-typed language like C or C++.

3.1 Identifying Structures to Transform

The compiler must be able to identify candidate data structures and ensure that the transformation will not cause a correct program to crash or exhibit errors. The challenge of transforming C and C++ programs is that they are not type safe. For splitting data structures we are not concerned about the type of the data. Rather we are concerned about the layout of the data because we are not changing how it is used, only where it is located.

The definition of structure layout used for MPADS is that the byte-level view of the structures must be the same for the structures to be considered the same layout. Formally, if two structures, s_1 and s_2, have the same number of fields and every field, f_i, in both structures at a given offset are the same length, then they are considered to have the same layout (*i.e.*, $lengthof(s_1.f_i) = lengthof(s_2.f_i)$ AND $offset(s_1.f_i) = offset(s_2.f_i)$).

For a transformation to be safe the candidate structures must have the same layout and all the pointer accesses to the candidate structures must be consistent with that layout. The reason why pointer accesses to the candidate structures must all have the same layout is because when we split the structures and reorganize the data in memory, we need to modify the pointer accesses by changing the address calculation for accesses to the fields. If two structures with different layouts are

in the same alias set, the structure cannot be transformed because modifying the field accesses is unsafe. Transforming the pointer may be unsafe because the pointer may access two different structures with a different layouts and the fields that were originally located at the same offset from the start of the structure are now in different locations.

Safe candidates are identified by analyzing the results of a pointer analysis and then combining the alias sets with information from the compiler's symbol table. The symbol table provides information about the layout of the object and the pointer analysis allows the compiler to determine which pointers are aliased.

The pointer analysis used is an inter-procedural Steensgaard's style analysis [61]. Steensgaard's pointer analysis is a field-sensitive, flow-insensitive and context-insensitive unification-based analysis. The analysis was chosen because it scales to large programs and is field-sensitive.

It is important for the alias analysis to be field-sensitive, because structures often contain many fields of different types and the coarse granularity of field-insensitivity would result in missed opportunities. If the pointer analysis did not differentiate between the fields, it is likely that every field in the object would be aliased with the pointer to the structure. When the pointer to the structure and the fields are aliased in a structure with different field types, the splitting opportunity would be unnecessarily abandoned.

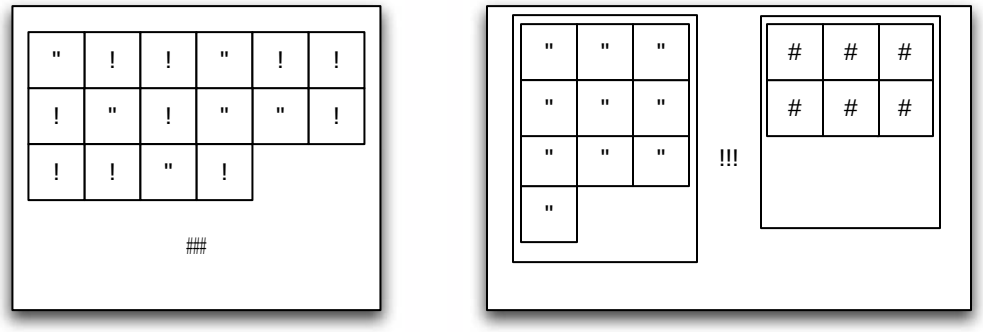
The pointer analysis provides more information then simply a check for safety, it is also used to determine in which pools the candidate structures should be allocated [29]. Using a unification-based alias analysis typically results in all of the pointers that access a particular data structure to be in the same alias set because the pointers used to access the data are be unified. Different alias sets represent different objects and are be allocated in different pools.

3.2 Memory Pooling

Memory pooling is the basis for the MPADS transformation and is an integral part of the transformation. Thus before describing how MPADS splits pointer-based structures, it is useful to describe the memory pooling mechanism used in MPADS.

When objects are allocated in the heap, it is possible that two different objects could be allocated in an interleaved pattern. With such allocation, objects from two different data structures could end up in the same cache line, resulting in poor locality and a polluted cache if the structures are transversed independently. Memory pooling groups similar objects together in pools to improve the spatial locality of the data.

For example, consider the interleaved allocation of two structures, A and B. If the allocations are performed using a standard allocation routine such as `malloc` then the data layout may look like Figure 3.1(a). However, if a pool allocation strategy is used the data layout will look like Figure 3.1(b). After pooling, the spatial locality between the type A structures has improved and the same is true for the type B structures. If structures A and B have the same layout and are aliased, it



(a) (b)

Figure 3.1: An example of two different structures, A and B, with interleaved heap allocations. (a) The allocations performed without pooling. (b) The allocations performed in the same order but using a memory pooling allocation method.

is possible that they will be allocated in the same pool.

3.2.1 Memory Allocation Library

The standard memory allocation functions do not provide support to allocate a group of objects together. Thus, a memory allocation library that can allocate similar objects in a pool must be created. The memory allocation calls are similar to those provided in the standard C library, only the memory allocated for each object will be in a pool that is managed by the memory allocation library.

The memory allocation functions take a *structure identifier* as a parameter. The structure identifier is used to tell the memory allocation library which allocations get grouped together.

APIs

The Memory Allocation Library must provide support for the common memory allocation calls found in the standard C library. This includes `malloc`, `calloc'` and `free`. Although `realloc` is a common memory allocation function, memory pooling and structure splitting is designed for cases where the data is allocated one item at a time. In all of the benchmarks tested, not one candidate used `realloc`. If more functionality is needed, the library can be extended to support more allocation functions in the future. Currently the supported APIs are:

- `void* pool_alloc(unsigned int struct_id, size_t struct_size, size_t pool_size);`
- `void* pool_calloc(unsigned int struct_id, size_t num_objs, size_t struct_size, size_t pool_size);`

- `void pool_free(void* ptr, unsigned int struct_id);`

Memory Pools

The Memory Allocation Library will manage a set of pools for each data structure. Distinct data structures are identified by the structure identifier, `struct_id`, that is passed into the allocation function. The pools have a fixed size, typically the same size as, or larger than, a page. Data is allocated contiguously within the pool until the pool is full. When the pool is full, another pool can be allocated and more memory can be allocated in this newly created pool.

Memory can be freed from the pools by using the Memory Allocation Library's `free` function. The freed objects are stored in a list and the memory can then be assigned to another allocation. When all of the memory in the pool is freed, the pool can be reclaimed.

Using multiple pools to store the data for each data structure allows MPADS to use only a small amount of additional memory while not limiting the framework to a fixed number of structures that can be allocated.

3.2.2 Compiler Transformation

The compiler can use the results of the pointer analysis to differentiate objects and allocate each structure in its own pool. To do this, the compiler must first determine which structures should be allocated in the same pool and then must replace the memory allocation calls with calls to the custom-made Memory Allocation Library.

The compiler starts the transformation by executing the pointer analysis and collecting all of the alias sets. The alias sets are then analyzed to determine which ones are valid candidates. Each alias set that is identified as a candidate is assigned a structure identifier. All of the structures that are accessed by the elements in the alias set will be allocated in the same pool. Essentially, each alias set represents a distinct object and will be allocated in its own pool.

The alias analysis has been modified to collect the allocation sites during the same pass that performs the analysis. The allocation sites are associated with an alias set and when two alias sets are unified the list of associated allocation sites is also unified. Allocation sites are identified by calls to `malloc`, `calloc`, `realloc`, `alloca`, `valloc`, `strdup`, `memcpy`, `memalign` and `posix_memalign`. Currently the transformation is only performed if the allocation site is a call to the `malloc` or `calloc` functions in the standard C library. If another allocation function is found the transformation is abandoned. The usage of other memory allocators suggests that the programmer may not be creating a standard pointer-based structure or that they are manually tuning their application and splitting may actually harm performance.

The compiler iterates through the list of allocation sites for each candidate and transforms the allocation to the corresponding call from the memory allocation library. The structure identifier from the alias set is passed as the `struct_id` parameter to the allocation function.

A list of the deallocation sites is also created during the alias analysis. The same process that was performed for the allocation sites is performed for deallocation sites. If the object is flagged to be transformed, the deallocation sites are also changed to use the corresponding functions in the Memory Allocation Library.

3.3 Structure Splitting

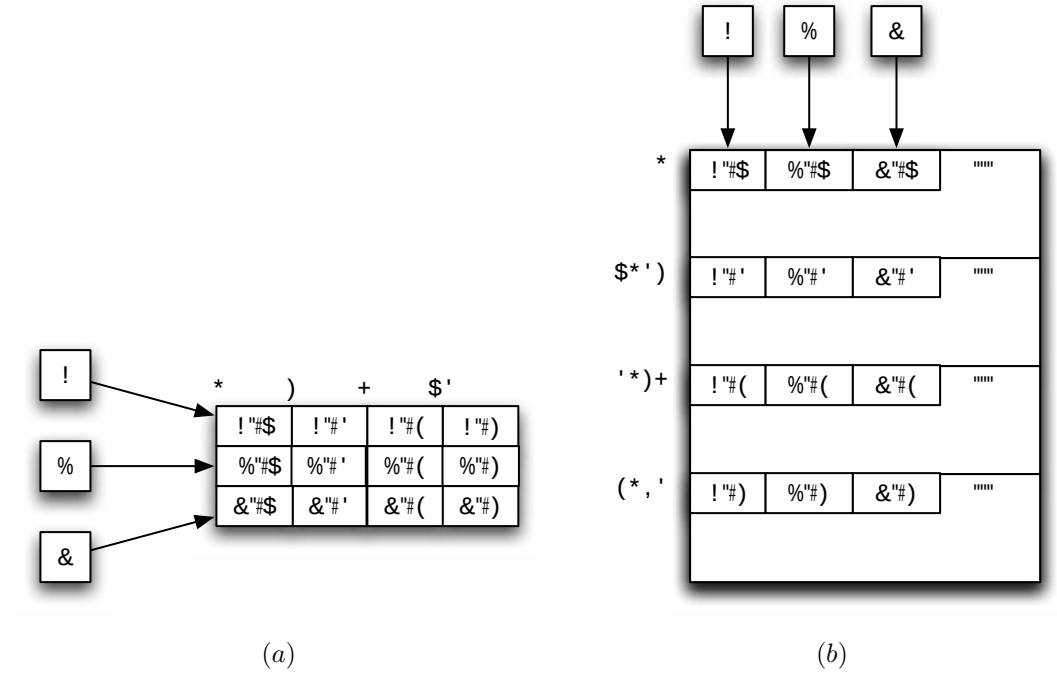
Memory pooling can increase the spatial locality of data by grouping or pooling similar structures together. Memory pooling works well when a traversal of a data structure accesses all or many of the fields in a node of the aggregate structure before moving to the next structure. However, quite often a traversal of an aggregate data structure may only access a small fraction of the fields in each structure. The un-accessed fields may share a cache line with the fields that were accessed, polluting the cache and using valuable memory bandwidth. Structure splitting is a technique that can address this problem.

When a structure is split, all of the similar fields in each structure are grouped together. For example, all of the first fields in a structure are allocated near each other, all of the second fields in a structure are allocated near each other and so on. The result is that the fields with the same offsets in different structures have good spatial locality.

Figure 3.2 gives an example of how three structures, A, B and C are allocated both with and without splitting. Each structure in the example has 4 fields, $f1$, $f2$, $f3$ and $f4$. When structures are allocated without splitting, like in Figure 3.2(a), the fields of each structure are located next to each other (*i.e.*, $A.f1$, $A.f2$, $A.f3$ and $A.f4$ have good spatial locality). When MPADS is used the data is organized as shown in Figure 3.2(b). In the split version, fields $A.f1$, $B.f1$ and $C.f1$ now have good spatial locality.

Splitting data structures can improve performance in several ways. If the traversal of a data structure only accesses a few fields of the structure, then splitting greatly increases locality, reduces the size of the working set, reduces the memory traffic and does not pollute the cache. Splitting the data also creates data streams that can be prefetched by the hardware prefetchers. Most hardware prefetch engines can support prefetching of multiple streams of data simultaneously.

For splitting structures there are three common methods: affinity-based splitting, frequency-based splitting and maximal splitting. Affinity-based splitting typically requires a profiling run to analyze and determine the affinity of the fields in a structure. Fields with a high affinity are grouped together and then the structure is broken into groups based on the field affinity. Frequency-based splitting also needs information about how often each field is accessed and this is typically obtained from a profile. The fields are grouped into frequently-accessed fields, known as *hot fields*, and infrequently-accessed fields, or *cold fields*. The structure is split to separate the hot and cold fields. Maximal splitting does not group any of the fields in a structure together, it completely separates every field in the structure. Figure 3.2(b) is an example of maximal splitting. No fields from a single



(a) (b)

Figure 3.2: An example of (a) three structures allocated without splitting and (b) three structures allocated with MPADS Maximal Splitting.

structure are grouped together. Each field from the structure is grouped with the fields from other structures at the same offset. A study of data splitting techniques has shown that maximal splitting can achieve the best or near best performance when compared with affinity- and frequency-based splitting [70]. MPADS uses maximal splitting.

When MPADS splits structures it removes the padding and eliminates any fields that are not referenced in the program. Padding is typically inserted by the compiler to minimize cache conflicts or because the instruction set architecture requires that fetches be aligned by a certain size to be efficiently executed. Moving the data essentially invalidates the reasoning for adding padding to a structure because the data will no longer be located at the original site. Thus the padding can be eliminated by the structure splitting transformation.

MPADS only works for splitting link based structures and is not designed for splitting arrays of structures. Zhao *et al.* [70] and Zhong *et al.* [71] have developed techniques for splitting arrays of structures.

A major challenge when splitting the structures is transforming the address computation. The new address computation must be efficient because memory references occur frequently. The additional overhead from adding instructions for the new address computation may not be offset by performance improvement from increasing data locality. To reduce the overhead of address calcu-

lation MPADS uses two different techniques for structure splitting depending on the layout of the structure.

3.3.1 Uniform Structure Splitting

If all of the fields in the structure are of the same length then the address computation is simpler and more efficient then the case where the fields are different lengths. When MPADS splits structures where every field in the structure is the same length it is referred to as uniform structure splitting.

In general, accessing a field via pointer p is calculated as: *(p + $offset$) where the $offset$ is the number of bytes from the start of the structure, typically a small value such as 4, 8, 30, *etc.* The transformed pointer dereference is still computed as *(p + $offset$) only now the $offset$ will be a much larger value. The new offset for field f_i using uniform splitting can be calculated as:

$$field\_i\_offset = field\_length * num\_structs\_per\_pool * i \tag{3.1}$$

The address calculation can be seen graphically in Figure 3.3. Since each of the fields are the same length the start of field f_i will be at the same distance from the start of its section of the pool as the pointer is away from the start of the pool. Therefore, adding up the size of all of the fields that can be stored in-between gives the offset that needs to be added to the pointer to access the correct field.

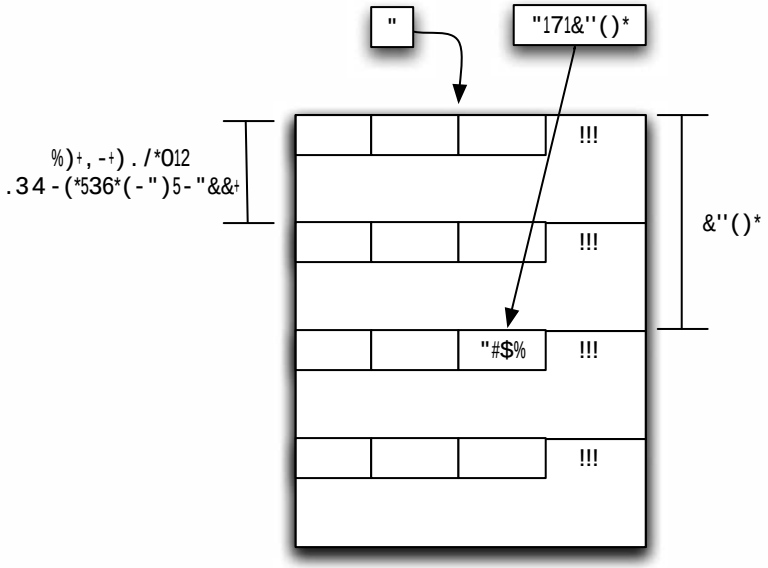


Figure 3.3: Pointer access with MPADS uniform maximal splitting.

If the target processor has a base plus offset addressing mode, there is likely a limited number of bits available to use for the offset. This will not limit the applicability of the method; either the pools could be made smaller or an additional add instruction could be used before the memory access.

3.3.2 Non-Uniform Structure Splitting

Non-uniform structure splitting refers to splitting structures that are comprised of fields that have different lengths. The fields are still allocated in the pool and split maximally, but because the field lengths are not the same, the address calculation is more complicated.

One drawback of using multiple pools for splitting structures with different size fields is that when a data field in a pool needs to be accessed, the start of the pool that it resides in must be identified in order for the index of the object to be computed.

For example, consider a pool that has several objects allocated in it, shown in Figure 3.4. Let the length of fields 1, 2, 3 and 4 be 2, 4, 4 and 8, respectively.

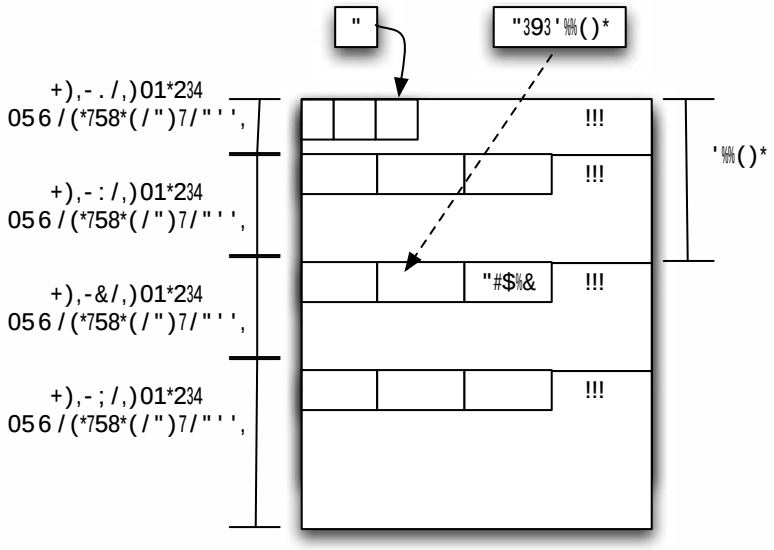


Figure 3.4: Example illustrating why the index in the pool must be known for non-uniform splitting.

Assume that there is a pointer p and the application wants to access field $f3$, $p->f3$. Assume that there are 100 structures in the pool and p points to the third object allocated. Calculating the offset similar to Equation 3.1 would give $(2 * 100) + (4 * 100) = 600$. However, this offset is actually 4 bytes short of the location that should be accessed. The dotted arrow in Figure 3.4 shows the data that would be accessed if the offset was 600 bytes. Thus, to access the correct location we need to know how many objects have been allocated in the pool before the structure referenced by the pointer.

It is useful to give an intuitive explanation of the new address calculation before describing the formal address calculation that can be found using Equations 3.2, 3.3 and 3.4. Figure 3.5 shows an example of a structure allocated in a pool. The basic idea behind the address calculation is to find distance of the field we are trying to access from the start of the pool. To do this we need to know how many other structures have been allocated in the pool up to and including the structure we are

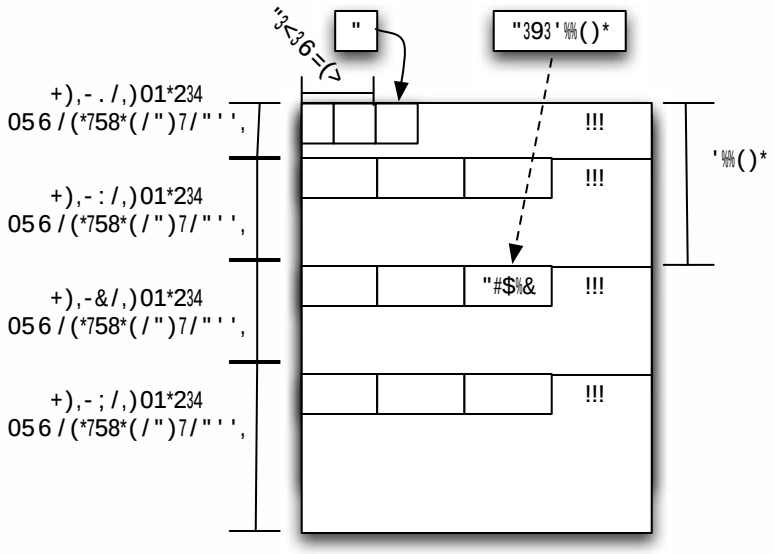


Figure 3.5: Pointer access with MPADS Non-Uniform Maximal Splitting

trying to access, we refer to this as the *index*. For example, the index of structure p in Figure 3.5 is 3. Now to find the distance from the start of the pool we need to add up the amount of the pool occupied for the fields before the field we are accessing. In addition, we need to add to the offset the space occupied by the number of fields allocated before the field we are trying to access. For example, in Figure 3.5 this would be two times the size of the third field in the structure. Finally, since we are adding an offset to p and not from the start of the pool we need to subtract the distance from p to the start of the pool.

```
UNSIGNED INT GENBITMASK(INT poolSize)
 1  unsigned int ret = 0
 2  int seen_a_1 = 0
 3  for i = 0 to sizeof(unsigned int) * 8
 4  do
 5     seen_a_1 = seen_a_1 || (poolSize & (0x01 << i)
 6     if seen_a_1 != 0
 7       then
 8            ret = BitwiseOR(ret, (0x01 << i))
 9
10  ret = BitwiseXOR(ret, 0xFFFFFFFF)
11  return ret
```

Figure 3.6: Algorithm to create the bit mask used for masking the pointers.

Using the runtime library to search for the start of the pool that the pointer belongs to and then returning its index in the pool would be expensive. To make address calculation inexpensive, MPADS aligns the memory allocated for the pools on boundaries that are multiples of the size of

the pool. If the pools are aligned then a simple binary *and* can be used to find the index of the object. The bit mask, $mask$, can be calculated using the function given in Figure 3.6. The bit mask is calculated at compile time and the constant that is returned is used in the address calculation. The index of the object can be found as follows:

$$index = \frac{p \,\&\, mask}{sizeof(f_1)} \tag{3.2}$$

$$num\_structs\_per\_pool = \frac{pool\_size}{\sum_i sizeof(f_i)} \tag{3.3}$$

$$field\_i\_offset = \begin{cases} 0 & \text{for} \quad i = 1 \\ \sum_{j=1}^{i-1} (sizeof(f_j) * num\_structs\_per\_pool) + & \text{for} \quad i > 1 \\ (sizeof(f_i) * index) - (p \& mask) & \end{cases} \tag{3.4}$$

The calculation of $field\_i\_offset$ from Equation 3.4 is shown graphically in Figure 3.5. All of the sub expressions in Equation 3.4 except for $index$ and p are known at compile time and can be folded to further reduce overhead. As well, $sizeof(f_i)$ is typically a power of 2 and the compiler can use a strength reduction to replace the division with a bit-shift operation. [1]

3.3.3 Changes to the Memory Allocation Library

Calls to the allocation and deallocation functions are still intercepted the same way as pool allocation but a slightly different function must be used. The allocation function for structure splitting still groups similar objects together, but the location and pattern of the memory for each field that is allocated differs from the pool allocation routines.

The main difference between the allocation function for structure splitting and pool allocation is that the pool allocation library returns addresses that are separated by the length of the structure while the allocation function for structure splitting returns addresses separated by the length of the first field. This is best explained with an example.

Assume that the pool size is 4k and we are allocating a 16-byte structure consisting of four 4-byte fields. A call to the pool allocation function returns memory address m. Thus the memory in locations $[m, m + 15]$ has been allocated. The memory in $[m, m + 3]$ has been reserved for the first field, $[m+4, m+7]$ for the second field and so forth. The second call to the pool allocation function will return $m + 16$ and the memory in $[m + 16, m + 31]$ has been allocated.

Using the same example, the structure splitting allocation function would return different address and reserve different areas in the pool. In this example there are 4 fields, each field will occupy one quarter of the pool or 1024 bytes. The first field of the first object allocated in the pool will be located at m and the function would return the address m. The second, third and fourth fields of the

[1] The $sizeof(f_i)$ is known at compile time because the length of each field in the structure must be known for the transformation to be identified as safe.

first object would be located at $m + 1024$, $m + 2048$ and $m + 3072$, respectively. The first field in the second object allocated in the pool will be located at $m + 4$ with the second, third and fourth fields of the second object being located at $m + 1028$, $m + 2052$ and $m + 3076$, respectively.

There are a few other minor changes that need to be made. For non-uniform splitting, pools must be aligned by the pool size and are allocated using the `posix_memalign` system call. As well, for both types of structure splitting we require that the pool size be known at compile time to reduce the cost of address computation. To make the memory library more flexible, the pool size can be passed in as a parameter. The compiler automatically generates this parameter and uses the same value for the address calculation.

The APIs for the splitting functions are the same as the pool allocation functions except that they also include parameters for the size of the first field in the structure, this must be known for the allocation function to return the correct address.

- `void* split_alloc(unsigned int struct_id, size_t first_field_size, size_t struct_size, size_t pool_size);`
- `void* split_calloc(unsigned int struct_id, size_t first_field_size, size_t num_objs, size_t struct_size, size_t pool_size);`
- `void split_free(void* ptr, unsigned int struct_id);`

Selecting the size for the pools is an important consideration for splitting and can vary depending on the application, input selection and target machine. Ideally, an oracle would allow the pool allocation library to determine exactly how much data will be allocated in each pool and the memory allocation library would only need to create one pool for each data structure. Unfortunately such an oracle does not exits so we propose a range of pool sizes.

If memory requirements are tight, as may be the case for embedded applications, the space overhead would be amortized out quicker using smaller pools. The minimum size for pool allocation and splitting to start yielding returns is the size of the cache line divided by the size of the smallest field in the structure times the size of the structure. This way each field in the pool fills at least one cache line. Using the same idea but replace cache line with virtual page is a practical upper limit for the suggested range.

3.3.4 Compiler Transformation

For splitting, the compiler identifies the candidate structures and intercepts the calls to the memory allocation and deallocation functions, the same as memory pooling only using the structure splitting allocation calls. Once the candidates have been identified, and the allocation functions changed, the compiler then needs to update all of the accesses to fields of the structure.

To change pointer accesses the compiler recursively traverses the parse-tree searching for a load of an address from the stack followed by a load or store, refereed to as an indirect load or store. Once

an indirect load or store is found the compiler determines which alias set the pointer is a member of. If the corresponding alias set has been flagged as a candidate for splitting, the address calculation used in the indirect load or store is changed to use either the uniform split or non-uniform splitting addressing described in Sections 3.3.1 and 3.3.2.

The offset for the first field in each structure is always 0 and can be accessed without a costly address computation. To try and improve the performance MPADS should put the most frequently accessed field at offset 0. Since profile information is not available, we assume that the recursive fields in most structures are accessed very frequently and MPADS makes that the first field. If there are multiple recursive fields MPADS arbitrarily picks one of them to be the first field.

3.4 Implementation in the IBM XL Compiler

The MPADS transformation is implemented in the Toronto Portable Optimizer (TPO) in the IBM XL compiler. MPADS required an inter-procedural pointer analysis to guarantee safety and thus it is a natural choice to implement MPADS in the TPO, which performs whole program optimization and analysis.

The TPO performs two passes over the program, the first pass collects information and analyzes the code while the second pass modifies the program. The MPADS framework could easily be integrated into the 2 passes that the TPO performs. On the first pass the pointer analysis is performed and candidate structures are identified and then on the second pass the candidate allocation sites and pointer de-references are modified.

MPADS added very little additional overhead to the compiler. The pointer analysis that MPADS uses is already performed by the TPO as part of the *Forma* array reshaping transformation [70]. Additionally, MPADS does not need to make any additional passes over the code because the pointer analysis provides enough information for the transformation process to be done locally, almost as though it is a peep-hole optimization.

Chapter 4

Performance on Micro Benchmarks

To determine the potential performance improvement provided by MPADS, two linked-list and one binary-tree micro benchmarks were created. MPADS was used to automatically reorganize the data structures in the micro-benchmarks and the performance results were collected. The micro-benchmarks showed that MPADS can significantly reduce cache misses and improve performance when compared with both pool allocation and the original version of the programs without data reorganization.

4.1 Experimental Setup

The benchmarks are evaluated on two different hardware architectures and are compiled with the IBM XL compiler at the highest optimization level, -O5.

The machines used for evaluation are a 1.7 GHz Power4 machine and a 1.9 GHz Power5 machine. The pertinent information about the memory hierarchy configuration of each machine is given in Table 4.1 and the memory latency for each level of the memory hierarchy is given in Table 4.2. [1] Both processors use a hardware prefetcher that can identify strided access patterns and automatically perform prefetching. An interesting architectural detail is that the L3 cache on the Power5 architecture is a victim cache [43].

All of the timing results are calculated by taking the smallest running time from 10 runs of the application. The performance metrics are gathered using the *tcount* tool that monitors the hardware counters and are gathered during a separate run so that it does not affect the timing results [65].

4.2 Micro Benchmarks

4.2.1 Linked List 1

The Linked List 1 benchmark creates a linked-list with 1.5 million nodes where each node contains five fields. The five fields in the structure are different sizes. The list is initialized and then traversed

[1] Information about the Power4 and Power5 machines was collected from various sources [19, 27, 41, 42].

	Power 4	Power 5
L1 Data Cache	32kb 2-way associative 128 byte cache line	32kb 4-way associative 128 byte cache line
L2 Cache	1.44Mb shared per chip 8-way associative 128 byte cache line	1.9Mb shared per chip 10-way associative 128 byte cache line
L3 Cache	32Mb per chip 8-way associative 512 byte cache line	32Mb per chip 12-way associative 256 byte lines
TLB	1024 entries 4-way set-associative	1024 entries 4-way set-associative

Table 4.1: Cache Configuration

	Power 4	Power 5
L1D Cache	1	4
L2 Cache	8 - 12	14
L3 Cache	118	80
Main Memory	250	351

Table 4.2: Memory Hierarchy Latency

1000 times. To simulate an interleaved allocation with another structure, 100 bytes are allocated between the list nodes.

There are two versions of the Linked List 1 benchmark, namely Linked List 1A and Linked List 1B. The data structure is the same but the traversal method is different. In the Linked List 1A program all of the fields in each node are accessed before the traversal continues to the next node. The list is traversed 1000 times in this fashion. Alternatively, for Linked List 1B a separate traversal is performed that will only access one field from the structure and the next pointer. Essentially this traversal only accesses one field in the node before moving to the next node. This is preformed 1000 times for each field before traversing the next field in the structure.

The two benchmarks were not designed to be compared. The reason that the two different access patterns were chosen is to test if structure splitting could improve the performance of both traversal patterns.

The source code for Linked List 1A and Linked List 1B can be found in listings B.1 and B.2 of Appendix B.

4.2.2 Linked List 2

Linked List 2 uses a large linked-list data structure with 2.1 million nodes where each node has 10 4-byte fields. Each field in the structure is traversed 100 times, *i.e.*, the structure is traversed 100 times accessing only the first field, then the structure is traversed 100 times accessing only the second field, and so forth.

This benchmark was tested both with and without data allocated between the list nodes. Data allocated between the list nodes is 40-bytes long and is refereed to as interleaved allocation.

4.2.3 Binary Tree

The Binary Tree benchmark is essentially the same as the Linked List 2 benchmark but a binary tree data structure replaces the linked list data structure used. The structure is traversed in a depth-first order accessing one field at a time. Each field in the structure is traversed 100 times.

Similar to Linked List 2, this benchmark is tested both with and without data allocated between the nodes of the list. The interleaved allocations are 40-bytes long.

4.3 Performance Results

For every one of the micro benchmarks, MPADS improved application performance compared with the baseline and memory pooling. MPADS improved performance because of the improved memory locality and better utilization of the memory hierarchy.

The speedup from using MPADS and memory pooling is shown in Figure 4.1. The baseline for comparison is a program compiled with the highest level of optimization, -O5. MPADS performed significantly better then memory pooling on both the Power4 and Power5 architectures. The speedup for MPADS ranged from a 1.47 fold to 12.36 fold improvement while pool allocation received a speed-up of 1.21 fold to 4.11 fold.

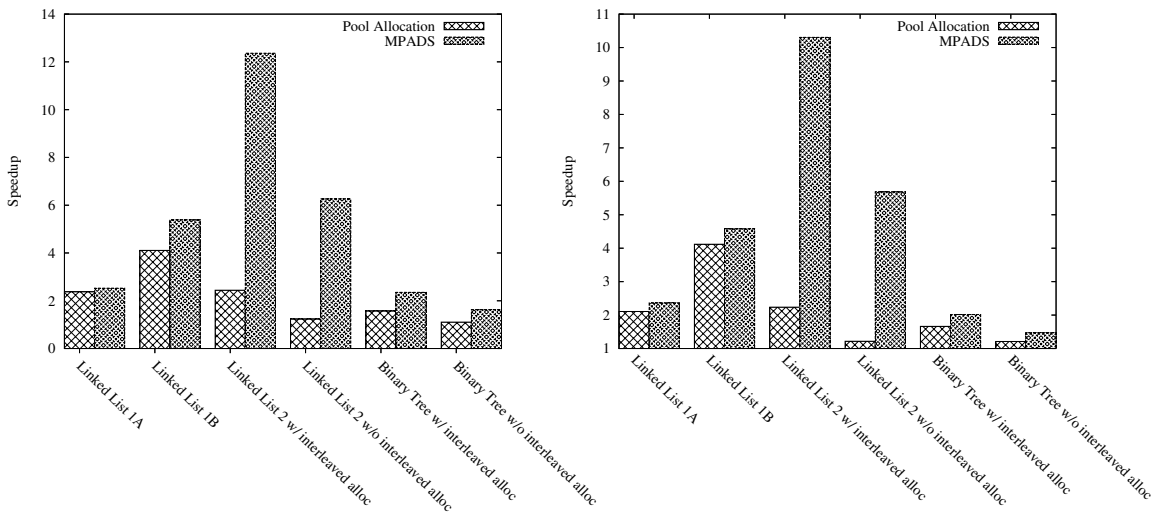


Figure 4.1: Speedup on a (a) Power4 and (b) Power5.

Both MPADS and memory pooling were able to successfully pool the interleaved allocations. In

Figure 4.1 it can be seen that the benchmarks with the interleaved allocations had a larger speedup then the benchmarks without the interleaved allocations. The reason that those benchmarks had greater performance improvements is because both MPADS and pool allocation placed the interleaved allocations into separate pools and kept the interleaved data from wasting memory bandwidth and polluting the cache.

Looking at the number of instructions executed and the average number of cycles per instruction (CPI) can help provide a picture of the impact that the transformation is having, these results are provided in Figures 4.2 and 4.3. As expected, the additional address calculation instructions in MPADS increased the number of instructions executed, but the improved data locality resulted in fewer stalls and a lower CPI. The largest reduction of CPI on the Power4 was for benchmark Linked List 2 with the interleaved allocations, dropping from 10.23 to 0.81. On the Power5, the largest reduction of CPI was for benchmark Linked List 1B dropping from 9.35 to 1.21.

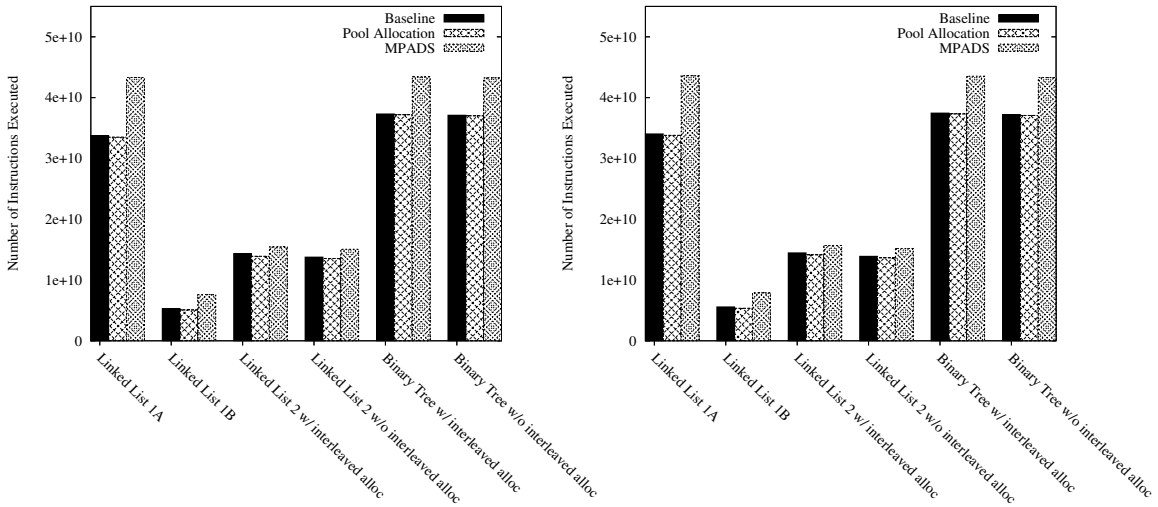


Figure 4.2: Dynamic instruction count on a (a) Power4 and (b) Power5.

A somewhat unexpected result was that pool allocation reduced the number of instructions executed. Examining the standard C library showed that comparatively, the pool allocation library calls are relatively efficient because they allocate large pools of memory and then just assign a small chunk of the pool for each allocation call. This helps to minimize the overhead of allocating memory and results in fewer instructions being executed then the baseline.

Even though the number of instructions executed increased with MPADS, the time required to execute the program decreased. The programs runs faster because the transformation reduced the size of the working set and was able to better utilize the hardware provided in the memory hierarchy.

The Translation Lookaside Buffer (TLB) is a small table in the CPU that is used to translate

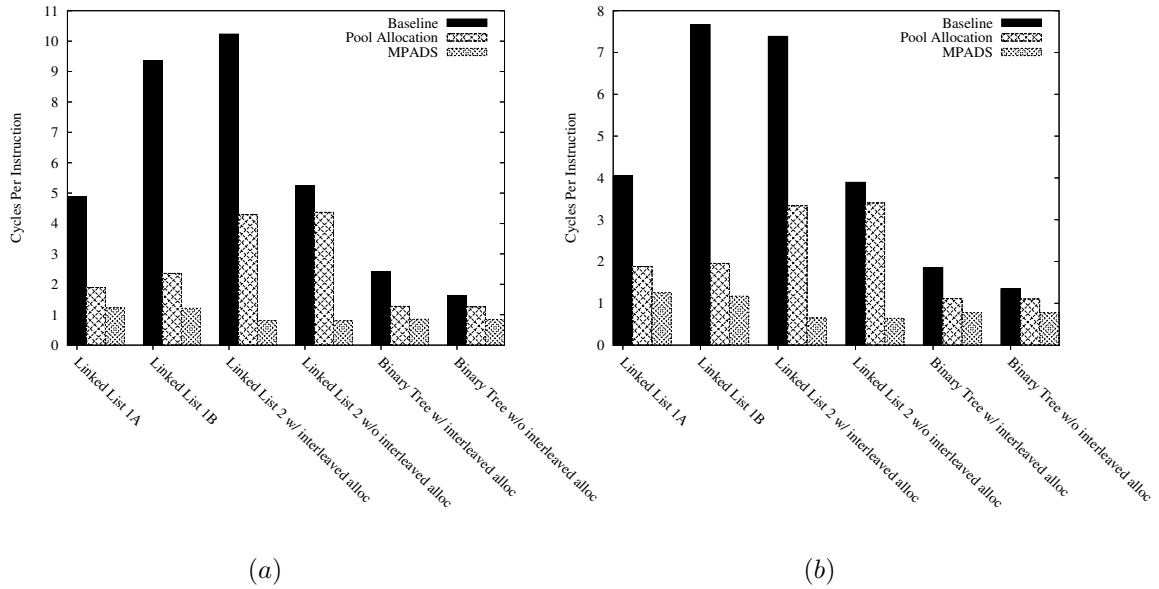


Figure 4.3: Cycles per instruction on a (a) Power4 and (b)Power5.

virtual addresses into physical addresses. If the working set is small enough, all of the address translations can be quickly handled using the TLB. If a virtual address is not found in the TLB then the CPU will typically trap to the operating system and walk the page tables to compute the address translation. TLB misses can be very expensive. The Power4 and 5 architectures use separate TLBs for data and instructions, abbreviated as the DTLB and ITLB, respectively. Since MPADS performs a data transformation we are only interested in the DTLB performance.

The number of DTLB misses are given in Figure 4.4. Every benchmark had the fewest number of DTLB misses when compiled with the MPADS optimization. Comparing MPADS with the baseline and pool allocation, it is clear that structure splitting had fewer TLB misses and this result is likely caused from reducing the size of the working set. For the benchmarks, MPADS reduced the number of DTLB misses by at least a factor of 4 from the baseline. Some benchmarks saw an improvement of more then 9 times fewer misses.

With memory accesses taking hundreds or even thousands of cycles, having the data in cache is critical to continue frequency scaling as a means of improving application performance. MPADS reduces the size of the working set and this reduction should allow more items to fit in cache. As well, data splitting can help prefetch the data implicitly, because the fields will be located on the same cache line. Data splitting will also help with explicit prefetching provided by hardware support because the data is organized into streams.

The L1D cache in the Power architectures is local to each processor core and there are separate data and instruction cache. The number of L1D misses are given in Figure 4.5. MPADS had fewer L1D misses then memory pooling on every benchmark except for Linked List 1A. Since Linked

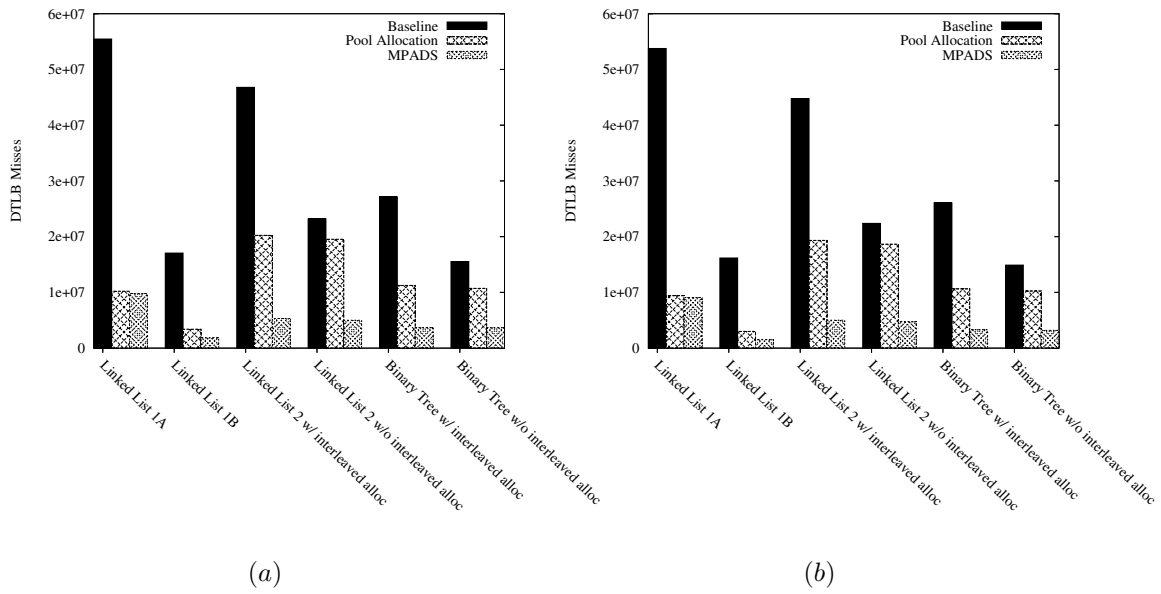


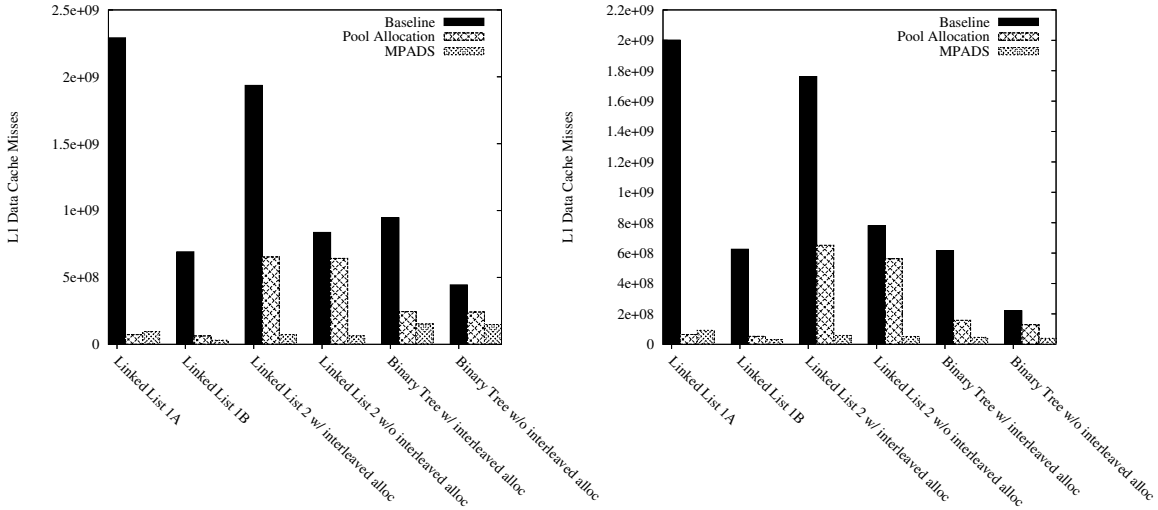
Figure 4.4: DTLB Misses on a (a) Power4 and (b) Power5.

List 1A accessed every field in the structure during each traversal its not surprising that structure splitting performed worse than memory pooling at the L1 level where the cache is small. It is worth noting that MPADS was only marginally worse than memory pooling on Linked List 1A and still reduced the number of L1D misses by 35 times from the baseline. The Linked List 2 and Binary Tree benchmarks compiled with MPADS had up to 17x fewer L1D misses then memory pooling and up to 35x fewer L1D misses then the baseline.

Like the L1 cache, there were far fewer L2 and L3 cache misses using MPADS. The number of L2 cache misses are given in Figure 4.6 and the number of L3 misses are given in Figure 4.7. For both the L2 and L3 caches the largest reduction of cache misses was on the Linked List 1A benchmark. For this benchmark, the differences between MPADS and memory pooling were small, but MPADS performed better than memory pooling on the Power4 machine while on the Power5 machine memory pooling performed better. For all of the other benchmarks MPADS performed better than memory pooling on both architectures. This is not surprising since structure splitting reduces the size of the working set and allow data to be prefetched more efficiently.

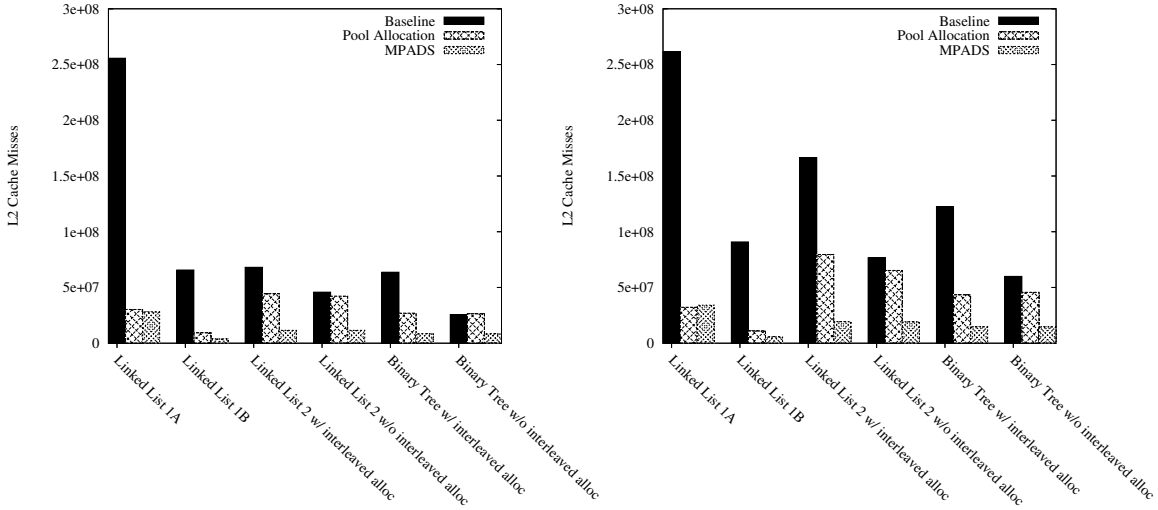
It may seem surprising that, for Linked List 1A, memory pooling performed slightly better then MPADS on the number of L1D misses yet MPADS had a larger speedup then memory pooling. This result can be explained by looking at the number of DTLB misses, MPADS has about 5% fewer DTLB misses than memory pooling. The reduction in the number of DTLB misses is enough to offset the increased number of cache misses.

The reason that Linked List 1A and 1B were created was to test if MPADS can perform better then memory pooling when (i) all of the fields in the structure are accessed contemporaneously



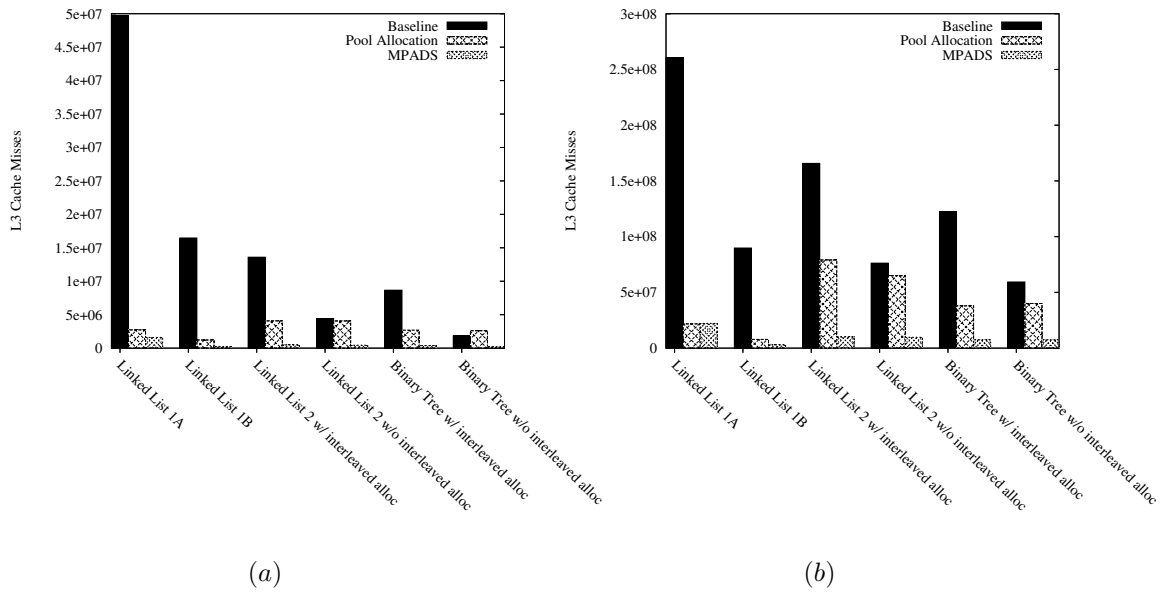
(a) (b)

Figure 4.5: L1D misses on a (a) Power4 and (b) Power5.



(a) (b)

Figure 4.6: L2 misses on a (a) Power4 and (b) Power5.



(a) (b)

Figure 4.7: L3 misses on a (a) Power4 and (b) Power5.

and (ii) each field is accessed individually. The results from the first micro benchmark indicate that MPADS has performance at least equal to pooling for both the A and B versions and can significantly outperform memory pooling when not all of the fields are accessed together.

Another interesting comparison to make is looking at the results for benchmarks with and without interleaved allocations. Once again, MPADS outperformed memory pooling in every benchmark. For the benchmarks without interleaved allocations, MPADS obtained a significantly larger reduction in the number of cache and DTLB misses compared with memory pooling. Data splitting significantly reduced the number of cache misses at all levels of the memory hierarchy for most of these benchmarks.

Memory pooling increased the number of L3 misses for the Power4 processor running the Binary Tree without interleaved allocations benchmark. However, pooling did not increase the number of L3 misses on the Power5 or on the benchmark with interleaved allocation. The increased number of misses for pool allocation is likely caused because there is no data allocated between the data and the baseline allocates everything contiguously. When pool allocation is applied the data is allocated in pools but the pools may not be contiguous. Thus there may be more space between the data and less locality leading to more cache misses. MPADS had fewer misses then both the baseline and the memory pooling.

MPADS performance improvement are not limited by the layout of the data structure. When the layout changed from a list to a tree, MPADS still outperformed memory pooling and obtained a speedup ranging from 1.46 to 2.35 times faster then the baseline, compared to a speed up ranging from 1.10 to 1.66 for memory pooling.

Chapter 5

Experimental Evaluation

After verifying that the MPADS performed as expected on the micro benchmarks, it must be evaluated on larger benchmark suites. For all of the larger benchmarks tested, MPADS outperforms memory pooling. For one of the benchmarks, MPADS cut the execution time in half, more then 27% better than memory pooling. However, the results for the rest of the benchmarks are mixed. Many potential opportunities were abandoned because the pointer analysis did not have enough precision and thus the transformation did not have as large an impact as expected. As well, the transformation caused one of the benchmarks, health, to have worse cache behavior and run 9% slower then the baseline.

5.1 Benchmarks

For the experiments, benchmarks from 3 sources were used, SPEC 2000, Olden [53] and LLU [74]. The Olden and LLU benchmarks were chosen because they have been used to evaluate many code transformations that aim to improve cache performance and because they contain pointer-based data structures [10, 29, 63]. The SPEC 2000 benchmarks were chosen because they are the *de facto* standard for performance measurement in the industry.

The benchmarks tested are comprised of C and C++ programs that use linked data structures. The size and layout of the data structures in the benchmarks varies. Some benchmarks use a standard linked list while others use structures such as a linked list of linked lists, or quad-trees. MPADS performs an analysis to identify candidates to split and should be able to split the structures regardless of the layout.

Optimization opportunities are not discovered in several of the benchmarks. Those benchmarks are not included in the results because a transformation was not performed on them and accordingly there is no change in their performance.

5.1.1 Missed Opportunities

For the Olden benchmarks, MPADS was unable to identify opportunities on half of the benchmarks, namely bisort, mst, perimeter, treeadd and voroni benchmarks. Thus, results for these benchmarks are not reported because they were not modified.

Opportunities are identified in only 5 of the SPEC 2000 benchmarks and the opportunities that were identified were responsible for referencing only a small fraction of each application's data. As a result the transformation did not have a measurable impact on any of the SPEC 2000 benchmarks. The number of opportunities identified and abandoned in each benchmark given in Table 5.1. The first two columns refer to the number of alias sets that were tagged as safe candidates for the transformation and the number of alias sets that were abandoned. The third column is the number of allocation sites that MPADS replaced with calls to the pool allocation library and the last column is the number of allocation calls in the program.

Although MPADS did not identify any opportunities in SPEC we believe that opportunities exist because Lattner and Adve's Data Structure Analysis (DSA) has successfully identified candidates in SPEC 2000 [32, 44]. Lattner and Adve's Data Structure Analysis uses a context-sensitive, field-sensitive, flow-insensitive unification-based pointer analysis that is more precise then the Steensgaard's style analysis used in MPADS. Unfortunately, neither DSA nor the results from DSA could be integrated into the implementation without a considerable amount of effort because the results are context-sensitive. Using the context-sensitive analysis would require modification of all of the function prototypes in the program to pass in a representation of the calling context so that data can be allocated in the correct pool depending on the calling context.

One of the SPEC benchmarks that may contain an optimization opportunity is *ammp*. Three of the main data structures in ammp contain dozens of fields and are very large. The structures *atoms*, *nodelist* and *atomlist* are 2208, 232 and 232 bytes each [1]. Only a small fraction of the fields in each structure are accessed during a traversal and this benchmark is an excellent candidate for MPADS. If this structure could be split it would likely result in a significant speedup. Unfortunately, this opportunity was not identified by MPADS.

The SPEC benchmark with the most opportunities identified was *gcc*. Twenty opportunities were identified in *gcc* but only 87 out of over 1500 allocation sites were modified and the allocation sites that were modified did not allocate any significant portion of data. The other opportunities were abandoned either because the alias sets contained different access patterns or the allocation site for an alias set could not be found. The pointer analysis not having enough precision is the reason that safe opportunities were abandoned.

	Num Pools Transformed	Num Pools Abandoned	Allocs Replaced	Allocs in Program
ammp	8	24	8	81
art	0	9	0	12
bzip	0	0	0	38
eon	0	3	0	152
gcc	20	60	87	1563
gzip	0	1	0	17
mcf	0	0	0	7
parser	0	0	0	144
perlbmk	1	3	2	125
twolf	2	24	7	203
vortex	0	0	0	16
vpr	1	37	36	293

Table 5.1: Limitations when transforming SPEC 2000

5.2 Results

For the experiments on the Olden, SPEC and LLU benchmarks, the experimental setup is the same as described in Chapter 4.1. The same machines and the same procedures were used to collect the results.

There was no noticeable increase in compilation time because the transformation used the results of a pointer analysis that the compiler already performs. During the transformation, the compiler did not have to do any additional passes over the code as it was integrated into the transformation phase already doing a pass over the code. The result was a very efficient implementation that did not significantly affect the compile time.

The MPADS transformation either outperformed or tied the performance of memory pooling on every benchmark. The speedup for each of the benchmarks after the transformations is given in Figure 5.1. Both memory pooling and MPADS had larger impacts on the Power4 processor than on the Power5. On the Power4, MPADS improved 5 benchmarks and memory pooling only improved 3. On the Power5 MPADS and memory pooling only improved 2 benchmarks but MPADS improved LLU by 27% more then memory pooling.

The CPI for the pool allocation and the baseline either stayed roughly the same or pool allocation had a sightly higher CPI, shown in Figure 5.2. For the MPADS optimized code the CPI was always smaller than memory pooling. Because of the address calculation, MPADS also executed more instructions than memory pooling on all of the benchmarks. Figure 5.3 shows the number of instructions executed.

The results for the Data Translation Lookaside Buffer (DTLB) shown in figure 5.5 are rather interesting. For health on the Power4 both memory pooling and MPADS caused more DTLB misses but on the Power5 the transformations caused fewer misses. Conversely, for tsp, MPADS caused fewer misses on the Power4 and more missed on the Power5. At first glance, these results appear

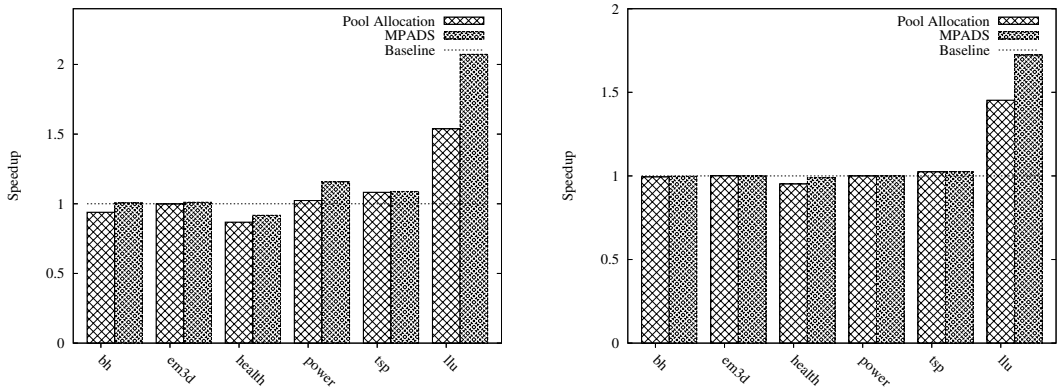


Figure 5.1: Speedup on a (a) Power4 and (b) Power5.

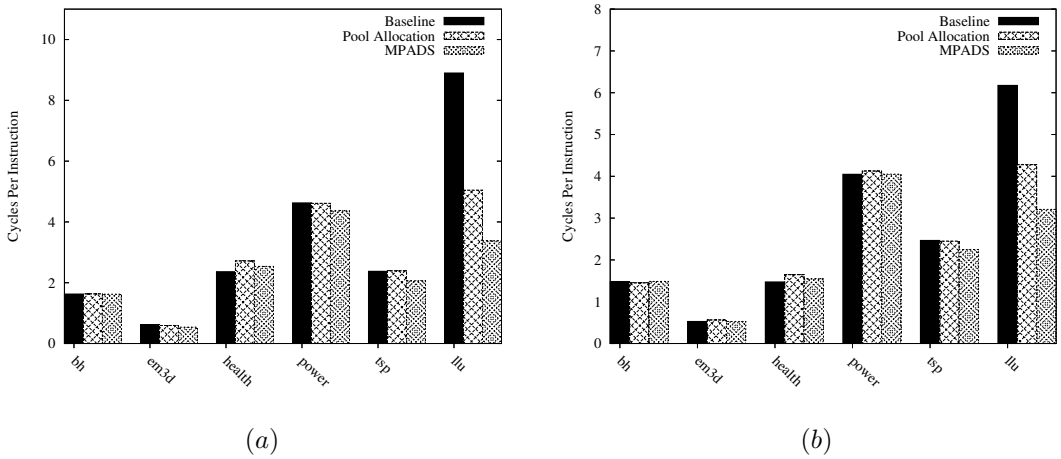


Figure 5.2: Cycles Per Instruction on a (a) Power4 and (b) Power5.

surprising considering that the TLBs on both processors have the same number of entries [41]. However, the Power5 architecture added a first level data translation table that contains a 128 entry fully associative array [60].

Perhaps most surprising is that for many of the benchmarks MPADS increased the number of L1D misses, shown in Figure 5.6, yet still obtained a speed up. Looking at the number of L2 and L3 misses in Figures 5.7 and 5.8 shows that MPADS decreased the number of misses at the lower levels of cache and these reductions outweighed the increases in L1D misses. The number of L1D cache misses increased most likely because memory pooling moved the two structures farther apart and the small L1D cache no longer contained both structures.

The benchmark health performed worse after structure splitting. Looking at the performance metrics the number of instructions, DTLB misses, L1D cache and L2 cache misses didn't change

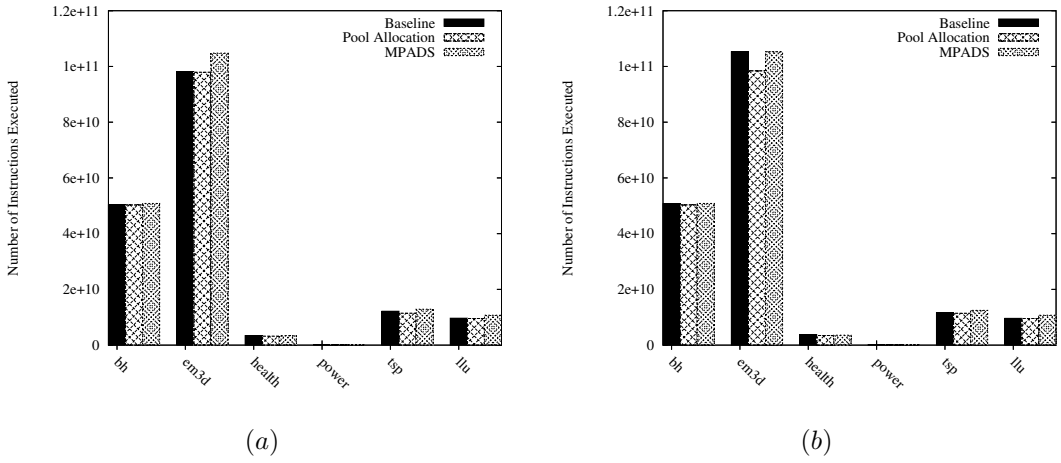


Figure 5.3: Dynamic instruction count on a (a) Power4 and (b) Power5.

very much. However, the MPADS transformation increased the number of L3 cache misses and the number of instructions executed resulting in the degraded performance. Health performed poorly because of the unique layout of the data structure used. Figure 5.4 from Zilles shows the data structure used in health to maintain the patient list [74]. Allocating the list structures and the patient structures in separate pools hurt performance because the patient structure is accessed via the list structure. Allocating them in separate pools moves them further away. It's interesting to note that splitting structures won some of the lost performance back and the largest slowdown for splitting was 9% while pool allocation slowed the benchmark down by 14%.

Data structures like the one used in health are commonly used by programmers and MPADS should be able to either improve them or abandon the opportunity. The slowdown appears to be caused by pool allocation separating the two lists. If both lists could be allocated in the same pool this would likely improve performance. Another alternative discussed in Chapter 7.1.2 is to abandon the transformation if the compiler can determine that it will degrade performance.

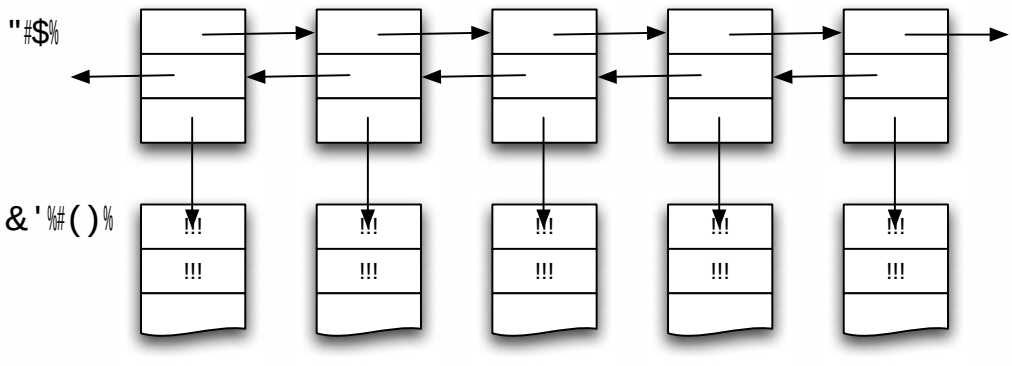


Figure 5.4: The data structure used in *health* to maintain the patient list from Zilles [74].

Benchmarks bh, em3d, power and tsp had much smaller performance improvements on the Power5 compared to the Power4. Looking at the results for the cache and DTLB misses shows that there was a much less significant reduction in misses on the Power5. As a result the number of cycles spent stalled only sightly decreased on the Power5 and most of that gain was eaten up by the overhead of the extra address calculation instructions.

The LLU benchmark simulates a linked list and was proposed as a replacement to the health benchmark that may not be representative of a typical linked list data structure [74]. This benchmark received the largest speedup from MPADS, 2.07 on the Power4 and 1.72 on the Power5. It's interesting that the number of L1 cache misses increased but the number of DTLB, L2 and L3 cache misses decreased. The L1 misses increased because the benchmark accesses many of the fields in the list nodes at the same time and the reorganization of the data caused poorer L1 cache performance. However, for the larger L2 and L3 caches the data reorganization allowed them to prefetch more data and significantly reduced the number of cache misses.

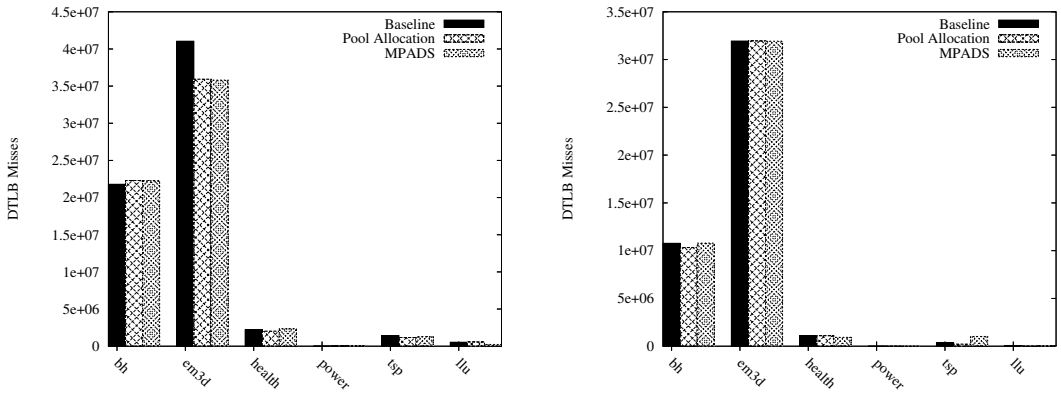
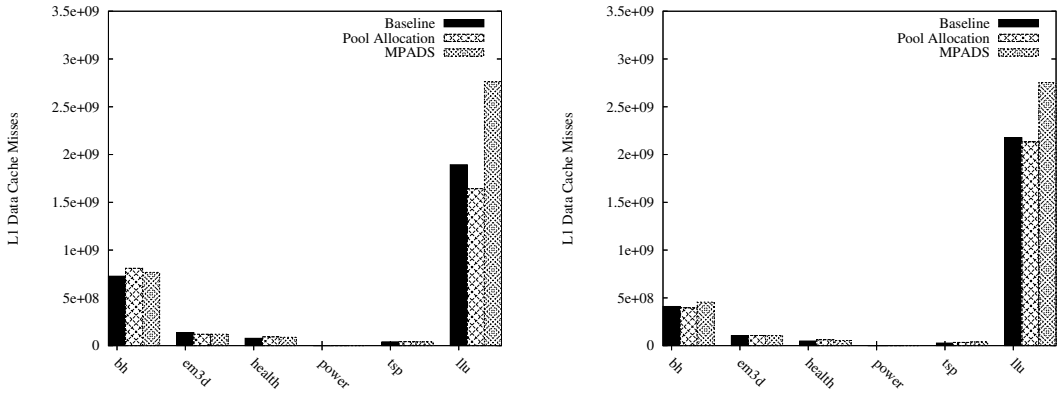
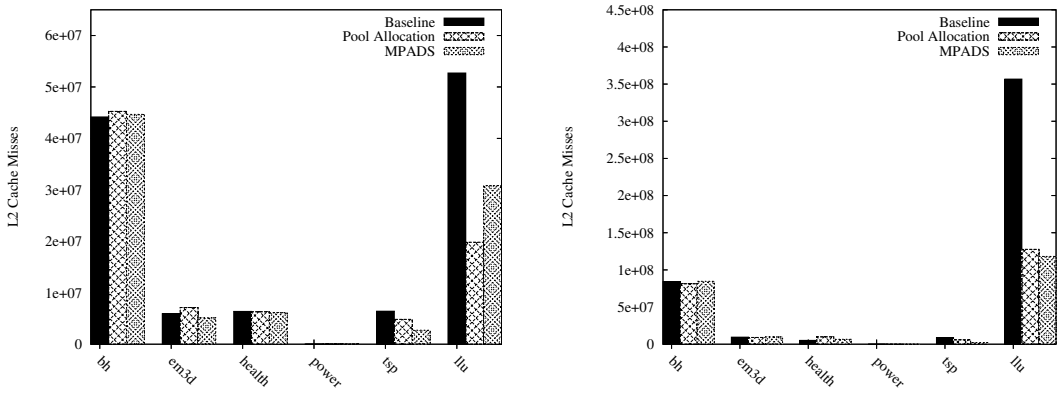


Figure 5.5: DTLB Misses on a (a) Power4 and (b) Power5.



(a) (b)

Figure 5.6: L1D Misses on a (a) Power4 and (b) Power5.



(a) (b)

Figure 5.7: L2 Misses on a (a) Power4 and (b) Power5.

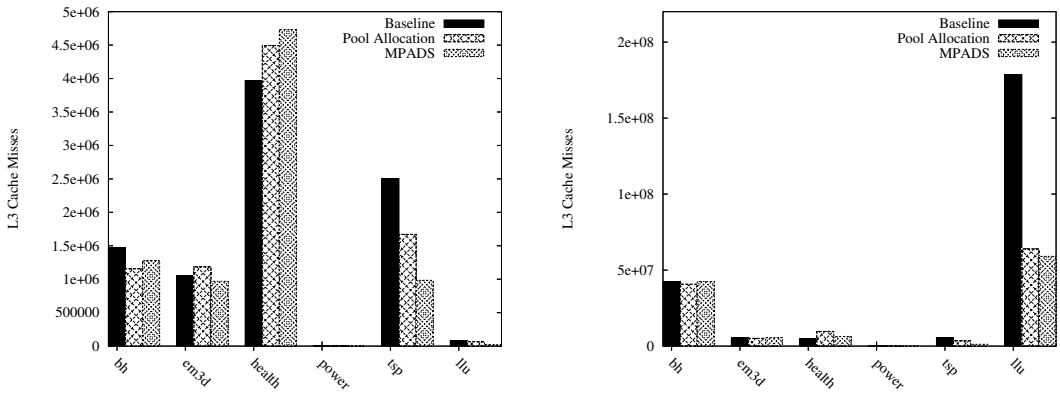


Figure 5.8: L3 Misses on a (a) Power4 and (b) Power5.

Chapter 6

Related Work

The focus of this thesis is data transformations for general purpose computing and the related work will be restricted to this domain. Many data transformations have been developed for scientific computing but they are generally not applicable to non-numeric applications and thus will not be covered.

6.1 Manual Data Transformations

Unless the performance of an algorithm is critical, most developers are not willing to incur the additional implementation and code maintenance costs that are required to perform locality-improving transformations. Researchers have acknowledged this and developed more general techniques and tools to help programmers modify their applications to reduce the running time. Unfortunately, many of the techniques and tools developed still require non-trivial programmer intervention.

Truong, Bodin and Seznec investigate two data transformations that they call field reorganization and instance interleaving [63]. Field reorganization groups fields that are referenced together into the same cache line. It reorganizes the order of the fields in the structures. Instance interleaving splits the data structure so that fields of different instances of a structure are grouped together. To support the instance interleaving technique, they develop a memory allocation library to change how the data is allocated. Instance interleaving is similar to MPADS structure splitting but it requires the programmers to manually modify their data structures by adding padding and replacing the memory allocation functions.

Chilimbi, Davidson and Larus modify the internal organization of fields in a data structure at compile time to improve the locality [9]. Two techniques are used to reorganize the fields in the data structures, *structure splitting* and *field reordering*. Structure splitting is used to increase the number of *hot* or frequently accessed fields found in a cache block. Structure splitting increases spatial locality and was shown to reduce cache miss rates by 10 to 27%, improving performance by 6 to 18%. Field reordering places fields with high temporal locality in the same cache block and moderately improve performance by 2 to 3 %. The algorithm used by Chilimbi, Davidson and Larus

requires profile information and the splitting is only safe for Java classes. As well, the structure splitting seperates the data into hot and cold fields and outlines the cold fields. A pointer to the structure with the cold fields is kept with the hot fields and everytime a cold field is accessed an aditional pointer dereference is required. MPADS structure splitting performs maximal splitting and although it requires more instructions for the address calculation it does not require an additional pointer dereference.

Chilimbi, Hill and Larus note that techniques for reducing memory latency have had limited success with pointer-based data structures and investigate methods to solve this problem [10]. They improve the locality of reference through two data placement techniques known as *clustering* and *coloring*. Clustering is a technique that places objects that have a high affinity in the same cache block. Coloring is a technique used to avoid conflict misses between heavily used memory locations. They create two methods named `ccmalloc` and `ccmorph` that use clustering and coloring techniques to create a cache conscious heap allocator. These techniques improved performance by up to 194% on some benchmarks but most benchmarks improved by 10 - 20%. The `ccmorph` function takes the first node in a list or a tree and an iterator as parameters. The `ccmorph` function reorganizes the data structure to be cache conscious by placing nodes and their children in the same cache block. This technique requires the API and functionality of iterators and the structure layout to be known by the `ccmorph` function. Transforming the data in the application requires significant effort on behalf of the programmer. Their techniques do not split data structures. Rather they allocate structures with high affinity together.

The novelty of the proposed techniques is that they try and address pointer-based applications with semi-automatic techniques. Although they still require significant programmer intervention they are a step in the right direction.

6.2 Automatic Data Transformations

The area of research most closely related to this work is the area of automatic data transformations. Automatic data transformations are appealing because they are transparent to the programmer and the compiler can often optimize programs better then the average programmer.

Finding a good data layout is a difficult problem. Even if we know the order that memory locations are accessed, the problem of organizing data in memory to minimize the number of cache misses can't be solved efficiently or even approximated very well unless P = NP [50]. Thus all of the proposed solutions are heuristics designed to improve the naive layout that is commonly used in production compilers.

Lattner and Adve developed one of the first fully automatic and safe data transformations to successfully transform dynamically allocated objects for general purpose programs written in type unsafe languages [29]. They created an analysis called Data Structure Analysis that is based on a context-sensitive pointer analysis. Their pool allocation automatically identifies safe candidates

to transform and allocates them in pools based on the objects that they were aliased with. The pool allocation idea forms the base for MPADS pool allocation and other structure splitting frameworks [24, 58].

Lattner and Adve used their Data Structure Analysis and pool allocation to safely compress pointers in linked data structures [30]. Their system reduces 64-bit pointers to 32-bit pointers by allowing pointers in the same pool to index other objects in that pool using an offset from the base of the pool. The system reduces the size of objects and allows more objects to fit in cache resulting in smaller working sets and improved application performance. MPADS attempts to reduces the size of the working set by reorganizing how data is placed in memory, opposed to modifying the size of the data.

Zhong *et al.* define a model to measure the closeness of references in a memory trace, the model is known as *reference affinity* [71]. Zhong *et al.* show how reference affinity can be used for structure splitting and array regrouping. Fields with high affinity are grouped together and then the structure is split into groups. Although they perform structure splitting in a compiler they assume that the language is type-safe and use programmer intervention to ensure that the transformation does not alter the semantics of the program. MPADS does not require a program trace and guarantees that the transformation is safe. As well, MPADS performs maximal splitting instead of affinity based splitting.

Zhao *et al.* implement *Forma*, a compilation framework to automatically and safely reshape single instantiated arrays [70]. Instead of using the affinity-based splitting used by Zhong *et al.*, Forma uses maximal splitting and shows that maximal splitting achieves best or near-best performance on the SPEC 2000 and Olden benchmarks.

Rabbah and Palem develop a completely automated data remapping technique that splits pointer-based structures [51]. Their system uses a trace of all the field accesses in the program to determine the field-access affinity. Field access affinity is used to decide which structures to split. The candidate structures are then split maximally similar to MPADS uniform structure splitting from Chapter 3.3.1. However, uniform splitting is the only splitting mechanism supported and the fields must be padded so that they are all the same length. If many fields in a structure are padded this can result in the data remapping polluting the cache more then in the original organization. MPADS only uses uniform structure splitting if all of the fields in the structure are the same size and thus does not pollute the cache with padding. Rabbah and Palem also use a field-insensitive pointer analysis whereas MPADS uses a more precise field-sensitive pointer analysis.

Shin *et al.* restructure the field layout for dynamically allocated objects [58]. Their field restructuring removes the padding in the structure, groups fields with high affinity together and performs affinity-based splitting. To determine which fields are accessed together the system uses profile information. Shin *et al.* describe the technique used to split the structures but do not describe how to integrate it into a compilation framework nor do they mention how they guarantee safety. Al-

though their technique is similar to the Non-Uniform Splitting technique described in Chapter 3.3.2, MPADS is different because it is designed to be safely integrated into a production compiler and performs maximal splitting. As well, MPADS supports uniform splitting to reduce the address calculation overhead when all of the fields in a structure are the same length. The numbers obtained by Shin *et al.*'s structure reshaping are slightly better then MPADS but they were obtained on a system with a higher clock speed and smaller caches.

Jeon, Shin and Han expand on Shin *et al.*'s previous work using structure splitting to reorganize objects allocated in the heap [24]. The major difference from their previous work is that this system is implemented in the CIL compiler framework and does not use profile information. The improvement to the field restructuring is the addition of a static analysis that uses regular expressions to represent the field access pattern. The regular expression can then be used to extract the access pattern and estimate the field affinity. Once again, affinity-based splitting is performed. The safety of their system is based on Lattner and Adve's observation that most pools are used in a type-consistent style [29]. Jeon, Shin and Han rely on their regular expressions to select candidates if the closure only contains fields from a single node and this will likely be enough for the majority of the cases. However, without a pointer analysis it is impossible to guarantee safety because fields can be accessed through pointers that may not be captured through their regular expression framework.

6.3 Prefetching

One of the first areas to attempt to reduce the stalls caused by high memory latency was prefetching. If the program could predict the future memory references then they could be loaded to cache before a fetch instruction is issued. Many of the techniques developed were software based but recently hardware prefetchers have been integrated into modern CPUs.

Luk and Mowry created three prefetching techniques for recursive data structures, greedy, history-pointer and data-linearizion prefetching [40]. Greedy prefetching was the only technique that was implemented into a compiler framework and didn't require programmer intervention. This automatic technique issued prefetches for the children of the node that was just fetched. For small memory latency this simple technique could improve application performance by as much as 45% but has little impact when memory latency is large. The two other techniques are designed to deal with larger memory latency but they required programmer intervention and were not automaticaly inserted by the compiler.

Rather then inserting software prefetches at compile time Chilimbi and Hirzel create a runtime system that profiles the memory accesses of data, determines hot streams and then inserts prefetch statements into the binary application at runtime [11]. The reduction of memory stalls was able to overcome the overhead of the profiling and analysis, resulting in performance improvements of up to 19% on some SPECint 2000 benchmarks.

Cahoon and McKinley use whole program analysis to identify profitable opportunities to in-

sert prefetching [6]. They implement their analysis, greedy-pointer prefetching and jump-pointer prefetching in a Java compiler. Jump-pointer prefetching is similar to Luk and Mowry's history-pointer prefetching, a jump-pointer field is added to each data structure. During the first traversal of the list the last n pointer accesses are recorded and inserted into the jump-pointer field in the structure. Thus each structure now has a pointer to the item that is referenced n items in the future. On subsequent traversals of the list the jump-pointer field is prefetched. Cahoon and McKinley note that although they obtain performance improvements as large as 48% a consistent improvement is hard to obtain.

Stoutchinin *et al.* present a prefetching algorithm to automatically prefetch data in a linked list [62]. The algorithm is based on the observation that list traversals regularly accesses memory that is separated by a constant distance, or stride. Their system automatically identifies pointer-chasing loops, determines if prefetching will be profitable, and inserts prefetch statements. Their system is integrated with the loop scheduling framework and only issues prefetches if the compiler estimates that there is enough memory bandwidth available and that the prefetches will not cause cache conflicts.

6.4 Cache-Conscious Algorithms

The performance bottleneck from high memory latency has led many researchers to look for solutions to address this problem. Some researchers are looking at addressing the fundamental problem via hardware-based solutions such as architectural changes or semiconductor development. Other researchers are attempting to mitigate the impact by modifying or re-writing their algorithms and creating tools to assist others with this task.

Applications with good data locality can perform substantially faster then those with poor locality [7, 57, 59]. Developers and researchers have started modifying algorithms in all areas of computing science and this has led to a new research area known as cache-conscious algorithms. This new area blends the knowledge of computer architecture with domain specific knowledge of a specific algorithm. Often, data structures are either split or reorganized to improve the data locality.

Rao and Ross changed the structure of B+ Trees used in main memory by reorganizing the data structure [52]. Rao and Ross realized that the child pointers in the tree were frequently accessed but many others fields stored near the child pointers were not frequently accessed. The modified program separated the child pointers from the nodes into their own data structure, improving the spatial locality for the pointer-chasing code. In their simulations, the modification reduced cache misses by approximately 50% when compared with a standard B+ Tree. Rao and Ross showed that splitting the B+ Trees reduced the size of the working set, improved data locality and reduced the number of cache misses.

Agrawal *et al.*'s *Apriori* frequent item-set mining algorithm is commonly used by businesses on large data sets where performance is critical [2]. *Apriori* is a perfect example of an algorithm that

has been extensively modified to reduce the running time. Ghoting *et al.* modified FPGrowth [20], considered the most efficient version of the Apriori algorithm, to replace the FP Tree data structure with a *tile-able cache-conscious prefix tree* that improves the spatial locality of the data [17]. The tile-able cache-conscious prefix tree is organized with paths from the root of the tree to the leafs being stored in tiles. Improving the cache behavior of this program improved the performance by a factor of 4.8x when compared with the previously best known implementation. Ghoting *et al.* then applied a similar technique to two other frequent item-set mining algorithms, Apriori and Genmax, and obtained speedups of 3.7 and 4.5 respectively. An important contribution from Agrawl *et al.* was that both single thread performance and hyper-threaded performance could be substantially improved by increasing data locality.

Chapter 7

Future Work

7.1 Opportunity Identification

7.1.1 Alias Analysis

As mentioned in the experiment section, chapter 5, Steensgaard's pointer analysis has enough precision to identify opportunities in the smaller benchmarks like LLU and Olden. However, the pointer analysis does not have enough precision to identify opportunities in the larger and more complex SPEC benchmarks.

Replacing the pointer analysis with a more precise analysis such as Lattner and Adve's Data Structure Analysis (DSA) could improve the performance of MPADS because it can transform more opportunities. However, such a change is not as simple as just plugging in a new alias analysis. DSA, for example, is context-sensitive and as such the alias sets are not valid at every point in the program. Thus some representation of the context must be known at runtime, this can be accomplished by either passing context strings as parameters or function cloning. The context must be known to ensure that the data is allocated in the correct pool and that the proper data access is performed.

A flow-sensitive analysis would also require a substantial re-engineering of the transformation because there is a different alias set at every point in the program.

7.1.2 Benefit Analysis

Although structure splitting can improve performance in many benchmarks there are some that it causes a performance degradation. It would be beneficial if the compiler could analyze each candidate to determine if structure splitting would be beneficial.

A static shape analysis like Ghiya and Hendren's [16] shape analysis could be used to help determine beneficial opportunities as well as a splitting plan. Only the objects that are likely to be improved with splitting will be identified as candidates.

Rubin, Bodik and Chilimbi acknowledge that finding an optimal layout is NP-hard and poorly approximately in polynomial time and create a profile-directed framework to search for good layouts in general purpose applications [54]. The framework captures a representative memory trace and

simulates different layouts using the captured trace. Their contribution is the ability to evaluate and combine optimization strategies without modifying the program and executing it. Such a system could be integrated into MPADS to determine which splitting strategy to perform. Such a framework could also be used to determine the benefits of splitting.

7.2 Affinity Analysis

The structure splitting framework can be extended by including affinity analysis to group fields with high affinity together where it may be profitable to do so. Other structure splitting frameworks have used profile-directed, trace-based, and static affinity analysis to perform affinity-based splitting [24, 51, 58, 71].

It is be possible to reduce the number of instructions executed by using uniform splitting instead of non-uniform splitting and padding the short fields to make every field in the structure the same length. Alternatively, field coalescing can be used to make structures with varying field lengths candidates for the uniform splitting technique. To make the field lengths the same, two or more fields could be coalesced based on affinity and field size. Field coalescing would eliminate much of the padding used in Rabbah and Palem's [51] structure splitting scheme.

Field affinity information can also be used with non-uniform splitting to group fields with high affinity together by performing affinity-based splitting.

Finally, using the affinity analysis the compiler could add padding or alignment to avoid cache conflicts. If two fields are often referenced together then it could try and align the memory to avoid conflict misses between these fields.

7.3 Other Techniques to Improve MPADS

There are several other techniques that can be implemented that may improve the MPADS splitting technique. Some are designed to help overcome some of the overhead and others are alternative ways to implement structure splitting.

The most interesting improvement to MPADS can eliminate some of the overhead of accessing fields in the Non-Uniform Splitting technique described in chapter 3.3.2. If the field being accessed is the same size as the first field in the structure then less addressing instructions are needed. The field can be accessed by summing up the sizes of the fields in-between in the pool. Formally, the new offset can be calculated as:

$$field\_i\_offset = \sum_{j=1}^{i-1} (sizeof(f_j) * num\_structs\_per\_pool) \tag{7.1}$$

Equation 7.1 can be calculated at compile time and thus would not increase the number of instructions executed. Exploiting this observation could significantly reduce the number of instruc-

tions executed and much of the overhead associated with Non-Uniform Splitting. However, it opens the door to a new question.

Since fields that have the same size as the first field in the pool can be accessed without additional overhead it would be beneficial to know the frequency each field in the structure is accessed to select the first field so that the number of additional addressing instructions is minimized during the execution of a program. However, finding the field access information typically requires profile information and may change depending on the input to the program

Another interesting idea that may improve the structure splitting is having the custom memory allocator return a pointer to a field in the middle of the structure rather then the first field in the structure. If the processor has a limited number of bits for addressing the base plus offset address calculations then this may allow the compiler to double the size of the pools without requiring an additional addition instruction.

The final idea revolves around another modification to the memory allocation library. Rather then using the typical 32 or 64-bit pointers returned by `malloc` it could use $n * 32$ or $n * 64$-bit pointers, where n is the number of distinct field sizes in each structure. Then all of the fields for each structure could be allocated in a pool with a uniform size. Each 32 or 64-bit chunk of the pointer could point to the first field of the pool with that distinct size. Thus Uniform Splitting could be applied to a non-uniform structure.

Chapter 8

Conclusions

A safe, automatic data structure splitting transformation, MPADS, is developed and implemented in a production compiler. The transformation performs two types of maximal splitting, uniform and non-uniform. The transformation uses a pointer analysis to automatically identify opportunities and to guarantee that the transformation is safe. MPADS does not require a program trace or profile information.

MPADS was not able to identify optimization opportunities in the SPEC 2000 benchmarks because of a lack of precision in the alias analysis. However, for the majority of Olden and LLU benchmarks, MPADS improves the application's performance. One of the benchmarks executed over 2x faster then the baseline after structure splitting. MPADS reduces the number of cycles required to execute each instruction by reducing stalls caused by cache misses but increases the number of instructions. This is a good trade-off for improving program performance by increasing clock frequency. Techniques similar to MPADS could allow hardware designers to continue reaping the benefits of frequency scaling.

Appendix A

Other Contributions

In addition to the MPADS structure splitting framework, several other contributions were developed throughout the course of my studies. This work was directed at improving static and feedback-based program analysis and have been implemented in widely used compilation frameworks such as McGill's Soot compilation framework [64], the Joeq / bddbddb program analysis framework [66, 67, 68] and the Open Research Compiler (ORC) [8]. The contributions are not directly related to the structure splitting work but it is possible that the structure splitting framework could be extended to use these contributions.

A.1 Using XBDDs and ZBDDs in Points-to Analysis

Points-to analysis is an important static analysis that is often needed for transforming programs written in languages with pointers or references. However, obtaining a precise analysis is infeasible for many large programs because of the space requirements of the analysis [39].

Binary Decision Diagrams (BDDs) can be used to efficiently represent the sets and relations that are commonly used in pointer analysis. BDD-based may-point-to and call-graph-construction analyzes were developed to improve the scalability of precise pointer analyzes [4, 38, 68, 72, 73]. To further reduce the storage requirements of pointer analysis, Lhoták, Curial and Amaral adapt XBDD and ZBDD data structures for points-to analysis [35, 36]. The XBDD and ZBDD data structures are a variation of BDDs that can represent the relations in points-to analysis more compactly then BDDs. Although both XBDDs and ZBDDs had been successfully used in other domains they had not been proposed for points-to analysis. To use ZBDDs in points-to analysis we had to develop the relation product operation for ZBDDs.

A BDD is a data structure used to represent a boolean function [5]. This function can be viewed as a set of bit vectors, namely those bit vectors that the function maps to true. A BDD represents such a function as a directed acyclic graph of nodes.

In this graph, a terminal node represents the constant `true` and another terminal node represents the constant `false`. Each non-terminal node, which specifies a BDD variable, has two outgoing

edges to other nodes, a `one` edge and a `zero` edge. The value of the function for a given valuation of the BDD variables is determined by a traversal starting at the root node of the BDD. At each node, the traversal follows either the one edge or the zero edge, depending on the value of the BDD variable associated with that node. Eventually, the traversal reaches a terminal node whose value indicates the value of the function.

Most implementations and algorithms that use BDDs are stored in their canonical form, called Reduced Ordered BDDs. An example of a binary function is given in figure A.1 (a) with the BDD shown in figure A.1 (b) and the ROBDD in figure A.1 (c).

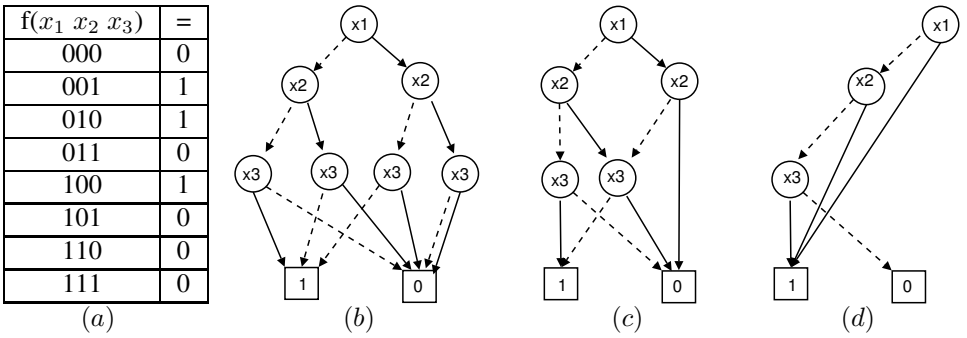


Figure A.1: The function f(x_1 x_2 x_3) (a), and the OBDD (b), ROBDD (c) and ZBDD (d) representing it. Solid edges represent one edges and dotted edges represent zero edges.

BDDs represent a binary function on n bits, thus there are exactly 2^n inputs to the function that will be mapped to either *true* or *false*. However, the number of elements in the domain to represent may not be a power of 2 and thus there are unused bit patterns. These unused inputs are represented in the BDDs and traditionally have been assigned to false, however, since the inputs will never be encountered they can be assigned to either true or false without affecting the accuracy or precision of the analysis. The size of BDDs can be reduced by eliminating these *don't-care* bit patterns. Lhoták, Curial and Amaral adapt two techniques to eliminate the don't care elements in pointer analysis, don't-care BDDs (XBDDs) and Zero-Suppressed BDDs (ZBDDs). Both techniques have been very effective at reducing BDDs size in applications such as circuit design, model checking, and verification but have not been used by the program analysis community.

Don't-care BDDs (XBDDs) refers to the technique of assigning true or false values to the unused inputs. Sauerhoff and Wegener showed that this problem is NP-hard [56] and thus, the *restrict operator*, a heuristic solution developed by Coudert and Madre, is used to efficiently find good solutions [12].

If more then the minimum number of required bits are used in the BDD then XBDDs were able to reduce the number of nodes in the BDD by 5 - 30%. However, when the programmer used the minimum number of bits necessary to create the BDD, XBDDs obtained only modest reductions of

a couple of percent.

A variation of BDDs, known as Zero-Suppressed BDDs (ZBDDs), is another promising alternative to eliminate the overhead of don't-care bit patterns [45, 46]. ZBDDs provide compact representations of sparse sets that are represented by a function with many zeros and few ones. A one-of-N encoding, rather than a binary encoding, is effective for ZBDDs. In a one-of-N encoding each element is assigned a bit pattern in which exactly one bit is set. The number of legal bit patterns is N, which need not be a power of two. Choosing N to be equal to the size of the domain of elements ensures that every legal bit pattern corresponds to some element, thereby eliminating don't-care bit patterns altogether. An example of a ZBDD is given in figure A.1 (d).

BDD-based points-to analysis requires the use of a relational product operation but there was no existing algorithm for ZBDDs. One of the contributions of this work is developing the relational product algorithm for ZBDDs. This algorithm allowed ZBDDs to be applied to points-to analysis.

ZBDDs showed a large potential for improving the larger sparser relations in pointer analysis. For context-sensitive analysis ZBDDs reduced the size of every relation used in the analysis. When represented with ZBDDs some relations had eight times fewer nodes. This reduction could allow otherwise infeasible analysis to be completed.

A.2 An Optimal Encoding to Represent a Single Set in an ROBDD

While investigating techniques to eliminate the overhead of unused bit patterns in BDDs; Lhoták, Curial and Amaral discovered an optimal solution to a related optimization problem [34].

The contribution is an optimal encoding for a set in a Reduced Ordered Binary Decision Diagram (ROBDD) when the number of elements in the set's domain is not a power of two. Unfortunately, the encoding cannot be used for BDD based points-to analysis because the domain of a BDD is created from the concatenation of several domains. Lhoták, Curial and Amaral discuss the more general optimization problem that must be solved for such a technique to be successful for BDD based points-to analysis [35].

A.3 Tree-Traversal Orientation Analysis

Data reorganization techniques such as structure splitting and pool allocation are designed to work for any shape of link-based data structure, be it a graph, dag or tree. It is possible that a data transformation could be optimized for specific data structure shapes and provide a greater performance improvement. Before specialized data transformations can be created, the compiler must be able to identify the shape of the data structure and analyze how it is accessed. Andrusky, Curial and Amaral create a feedback-directed analysis, named Tree-Traversal Orientation Analysis, that can determine if the traversal of a data structure is a list or tree given that a shape analysis can identify the structure as a directed acyclic graph. [25]. If the analysis identifies that the structure is a tree then it will also

identify if the traversal orientation is breadth-first, depth-first or a combination of those.

The analysis was implemented in the Open Research Compiler (ORC) and tested on several micro-benchmarks as well as the Olden benchmark suite. For all of the benchmarks tested, the analysis was able to correctly identify the orientation of the traversal.

Appendix B

Micro Benchmark Code Listing

B.1 Linked List 1A

```
#include <stdlib.h>
#include <stdio.h>

struct student_rds{
  unsigned short year_born;
  unsigned int sid;
  int data1;
  double data;
  struct student_rds *next;
};

static int NUM_STUDENTS = 1500000;

struct student_rds* initStudentListLoop();
void traverseStudentList(struct student_rds* list);


int main(int argc, char* argv[]){

  struct student_rds *student_list;

  student_list = initStudentListLoop();

  for(int i = 0; i < 1000; i++)
    traverseStudentList(student_list);

  free(student_list);

  return 1;
}

void traverseStudentList(struct student_rds* list)
{
  while(list->next != 0){
    (list->sid)++;
```

```
    (list->year_born)--;
    list->data = list->data + list->data1;
    list = list->next;
  }
}

struct student_rds* initStudentListLoop()
{
  struct student_rds *rtn, *list;
  char *chars;

  rtn = (struct student_rds*)malloc(sizeof(struct student_rds));
  rtn->next = 0;
  list = rtn;



  for(int i = 1; i < NUM_STUDENTS; i++){
    list->next =
          (struct student_rds*)malloc(sizeof(struct student_rds));

    chars = malloc(100); //malloc memory in-between structures

    if(i % 2)
      list->year_born = 3;
    else
      list->year_born = 4;
    list->data = 1234 + i;
    list->data1 = i \% 33;
    list->sid = i;
    list = list->next;
    list->next = 0;
  }

  return rtn;
}
```

B.2 Linked List 1B

```
#include <stdlib.h>
#include <stdio.h>

struct student_rds{
  unsigned short year_born;
  unsigned int sid;
  int data1;
  double data;
  struct student_rds *next;
};

static int NUM_STUDENTS = 1500000;
```

```
struct student_rds* initStudentListLoop();
void traverseStudentList1(struct student_rds* list);
void traverseStudentList2(struct student_rds* list);
void traverseStudentList3(struct student_rds* list);


int main(int argc, char* argv[]){

  int iterations = 100;

  struct student_rds *student_list;

  student_list = initStudentListLoop();

  for(int i = 0; i < iterations; i++)
    traverseStudentList1(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList2(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList3(student_list);

  free(student_list);

  return 1;
}

void traverseStudentList1(struct student_rds* list)
{
  while(list->next != 0){
    (list->sid)++;
    list = list->next;
  }
}

void traverseStudentList2(struct student_rds* list)
{
  while(list->next != 0){
    (list->year_born)--;
    list = list->next;
  }
}

void traverseStudentList3(struct student_rds* list)
{
  while(list->next != 0){
    list->data = list->data + list->data1;
    list = list->next;
  }
}

struct student_rds* initStudentListLoop()
```

```
{
  struct student_rds *rtn, *list;
  char *chars;

  rtn = (struct student_rds*)malloc(sizeof(struct student_rds));
  rtn->next = 0;
  list = rtn;

  for(int i = 1; i < NUM_STUDENTS; i++){
    list->next =
        (struct student_rds*)malloc(sizeof(struct student_rds));

    chars = malloc(100); //malloc memory in between structures

    if(i % 2)
      list->year_born = 3;
    else
      list->year_born = 4;
    list->data = 1234 + i;
    list->data1 = i \% 33;
    list->sid = i;
    list = list->next;
    list->next = 0;
  }

  return rtn;
}
```

B.3 Linked List 2

```
#include <stdlib.h>
#include <stdio.h>

struct student_rds{
  int data1;
  int data2;
  int data3;
  int data4;
  int data5;
  int data6;
  int data7;
  int data8;
  int data9;
  struct student_rds *next;
};

static int NUM_STUDENTS = 2100000;


struct student_rds* initStudentListLoop();
void traverseStudentList1(struct student_rds* list);
```

```
void traverseStudentList2(struct student_rds* list);
void traverseStudentList3(struct student_rds* list);
void traverseStudentList4(struct student_rds* list);
void traverseStudentList5(struct student_rds* list);
void traverseStudentList6(struct student_rds* list);
void traverseStudentList7(struct student_rds* list);
void traverseStudentList8(struct student_rds* list);
void traverseStudentList9(struct student_rds* list);

int main(int argc, char* argv[]){


  int iterations = 100;

  struct student_rds *student_list;

  student_list = initStudentListLoop();

  for(int i = 0; i < iterations; i++)
    traverseStudentList1(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList2(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList3(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList4(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList5(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList6(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList7(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList8(student_list);

  for(int i = 0; i < iterations; i++)
    traverseStudentList9(student_list);

  free(student_list);

  return 1;
}

void traverseStudentList1(struct student_rds* list)
{
  while(list->next != 0){
    list->data1++;
```

```
    list = list->next;
  }
}

void traverseStudentList2(struct student_rds* list)
{
  while(list->next != 0){
    list->data2++;
    list = list->next;
  }
}
void traverseStudentList3(struct student_rds* list)
{
  while(list->next != 0){
    list->data3++;
    list = list->next;
  }
}
void traverseStudentList4(struct student_rds* list)
{
  while(list->next != 0){
    list->data4++;
    list = list->next;
  }
}
void traverseStudentList5(struct student_rds* list)
{
  while(list->next != 0){
    list->data5++;
    list = list->next;
  }
}
void traverseStudentList6(struct student_rds* list)
{
  while(list->next != 0){
    list->data6++;
    list = list->next;
  }
}
void traverseStudentList7(struct student_rds* list)
{
  while(list->next != 0){
    list->data7++;
    list = list->next;
  }
}
void traverseStudentList8(struct student_rds* list)
{
  while(list->next != 0){
    list->data8++;
  }
}
void traverseStudentList9(struct student_rds* list)
{
```

```
  while(list->next != 0){
    list->data9++;
    list = list->next;
  }
}

struct student_rds* initStudentListLoop()
{
  struct student_rds *rtn, *list;
  char *chars;

  rtn = (struct student_rds*)malloc(sizeof(struct student_rds));
  rtn->next = 0;
  list = rtn;

  for(int i = 1; i < NUM_STUDENTS; i++){
    list->next =
          (struct student_rds*)malloc(sizeof(struct student_rds));

    chars = malloc(40); //malloc memory in between structures

    if(i % 2)
      list->data1 = 3;
    else
      list->data1 = 4;

    list->data2 = 1234 + i;
    list->data3 = i % 53;
    list->data4 = 123 + i;
    list->data5 = i % 99;
    list->data6 = 234 + i;
    list->data7 = i % 4;
    list->data8 = 1 + i;
    list->data9 = i % 39;
    list = list->next;
    list->next = 0;
  }

  return rtn;
}
```

B.4 Binary Tree

```
#include <stdlib.h>
#include <stdio.h>

struct student{
  int data1;
  int data2;
  int data3;
  int data4;
```

```
  int data5;
  int data6;
  int data7;
  int data8;
  int data9;
  struct student *left;
  struct student *right;
};

static int NUM_STUDENTS = 1000000;

struct student* initStudentTree(int num);
void traverseStudentTree1(struct student* tree);
void traverseStudentTree2(struct student* tree);
void traverseStudentTree3(struct student* tree);
void traverseStudentTree4(struct student* tree);
void traverseStudentTree5(struct student* tree);
void traverseStudentTree6(struct student* tree);
void traverseStudentTree7(struct student* tree);
void traverseStudentTree8(struct student* tree);
void traverseStudentTree9(struct student* tree);

int main(int argc, char* argv[]){

  int iterations = 100;

  struct student *student_tree;

  student_tree = initStudentTree(NUM_STUDENTS);

  for(int i = 0; i < iterations; i++)
    traverseStudentTree1(student_tree);

  for(int i = 0; i < iterations; i++)
    traverseStudentTree2(student_tree);

  for(int i = 0; i < iterations; i++)
    traverseStudentTree3(student_tree);

  for(int i = 0; i < iterations; i++)
    traverseStudentTree4(student_tree);

  for(int i = 0; i < iterations; i++)
    traverseStudentTree5(student_tree);

  for(int i = 0; i < iterations; i++)
    traverseStudentTree6(student_tree);

  for(int i = 0; i < iterations; i++)
    traverseStudentTree7(student_tree);

  for(int i = 0; i < iterations; i++)
    traverseStudentTree8(student_tree);
```

```
  for(int i = 0; i < iterations; i++)
    traverseStudentTree9(student_tree);

  free(student_tree);

  return 1;
}

void traverseStudentTree1(struct student* tree)
{
  if(tree == 0)
    return;

  tree->data1++;

  traverseStudentTree1(tree->left);
  traverseStudentTree1(tree->right);
}

void traverseStudentTree2(struct student* tree)
{
  if(tree == 0)
    return;

  tree->data2++;

  traverseStudentTree2(tree->left);
  traverseStudentTree2(tree->right);
}

void traverseStudentTree3(struct student* tree)
{
  if(tree == 0)
    return;

  tree->data3++;

  traverseStudentTree3(tree->left);
  traverseStudentTree3(tree->right);
}

void traverseStudentTree4(struct student* tree)
{
  if(tree == 0)
    return;

  tree->data4++;

  traverseStudentTree4(tree->left);
  traverseStudentTree4(tree->right);
}

void traverseStudentTree5(struct student* tree)
{
```

```
  if(tree == 0)
    return;

  tree->data5++;

  traverseStudentTree5(tree->left);
  traverseStudentTree5(tree->right);
}

void traverseStudentTree6(struct student* tree)
{
  if(tree == 0)
    return;

  tree->data6++;

  traverseStudentTree6(tree->left);
  traverseStudentTree6(tree->right);
}

void traverseStudentTree7(struct student* tree)
{
  if(tree == 0)
    return;

  tree->data7++;

  traverseStudentTree7(tree->left);
  traverseStudentTree7(tree->right);
}

void traverseStudentTree8(struct student* tree)
{
  if(tree == 0)
    return;

  tree->data8++;

  traverseStudentTree8(tree->left);
  traverseStudentTree8(tree->right);
}

void traverseStudentTree9(struct student* tree)
{
  if(tree == 0)
    return;

  tree->data9++;

  traverseStudentTree9(tree->left);
  traverseStudentTree9(tree->right);
}
```

```
struct student* initStudentTree(int num)
{
  struct student *tree;
  char  *chars;

  if(num <= 0)
    return 0;

  tree = (struct student*)malloc(sizeof(struct student));

  chars = malloc(40); //malloc memory in between structures

  if(num % 2)
    tree->data1 = 3;
  else
    tree->data1 = 4;

  tree->data2 = 1234 + num;
  tree->data3 = num % 53;
  tree->data4 = 123 + num;
  tree->data5 = num % 99;
  tree->data6 = 234 + num;
  tree->data7 = num % 4;
  tree->data8 = 1 + num;
  tree->data9 = num % 39;

  tree->left = initStudentTree(num/2);
  tree->right = initStudentTree(num/2);

  return tree;
}
```

Appendix C

Trademarks

IBM, XL Fortran, XL C, XL C/C++, XL UPC, POWER4 and POWER5 are trademarks of International Business Machines Corporation in the United Sates, other countries, or both.

Bibliography

[1] K. K. Agaram, S. W. Keckler, C. Lin, and K. S. McKinley. The memory behavior of data structures in C - SPEC CPU2000 benchmarks. In *2006 SPEC Benchmark Workshop*, January 2006.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 207–216, New York, NY, USA, 1993. ACM Press.

[3] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelickak. The landscape of parallel computing research: A view from Berkeley. Technical report, UC BERKELEY, 2006.

[4] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Inplementation*, pages 103–114, 2003.

[5] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.

[6] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT '01: Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, Washington, DC, USA, 2001. IEEE Computer Society.

[7] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 139–149, 1998.

[8] S. Chan, Z. H. Du, R. Ju, and C. Y. Wu. Web page: Open research compiler - aurora. http://sourceforge.net/projects/ipf-orc.

[9] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 13–24, New York, NY, USA, 1999. ACM Press.

[10] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, pages 1–12, New York, NY, USA, 1999. ACM Press.

[11] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 conference on Programming Language Design and Implementation*, page 1990209, Berlin, Germany, 2002.

[12] O. Coudert and J. C. Madre. A unified framework for the formal verification of sequential circuits. In *IEEE International Conference on Computer-Aided Design. ICCAD-90*, pages 126–129, Santa Clara, CA, USA, Nov. 1990.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[14] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 229–241, New York, NY, USA, 1999. ACM Press.

[15] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[16] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages (POPL)*, pages 1–15, St. Petersburg, Florida, January 1996.

[17] A. Ghoting, G. Buehrer, S. Parthasarathy, D. Kim, A. Nguyen, Y. Chen, and P. Dubey. Cache-conscious frequent pattern mining on a modern processor. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 577–588. VLDB Endowment, 2005.

[18] J. Gosling. Java intermediate bytecodes: Acm sigplan workshop on intermediate representations (ir'95). *SIGPLAN Not.*, 30(3):111–118, 1993.

[19] C. Grassl. Optimization and tuning on POWER4 systems. The IBM System Scientific Computing User Group: http://www.spscicomp.org/ScicomP5/Presentations/Tutorial/Daresbury.POWER4.Tuning.tut.pdf, May 2002.

[20] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 53 – 87. ACM Press, 05 2000.

[21] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 290–299, New York, NY, USA, 2007. ACM Press.

[22] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 97–105, New York, NY, USA, 1998. ACM Press.

[23] M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Sci. Comput. Program.*, 39(1):31–55, 2001.

[24] J. Jeon, K. Shin, and H. Han. Layout transformations for heap objects using static access patterns. In *Compiler Construction*, pages 187–201, March 2007.

[25] S. Curial K. Andrusky and J. N. Amaral. Tree-traversal orientation analysis. In *19th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 220 – 235, New Orleans, Louisiana, November 2006.

[26] M. Kandemir and I. Kadayif. Compiler-directed selection of dynamic memory layouts. In *CODES '01: Proceedings of the ninth international symposium on Hardware/software codesign*, pages 219–224, New York, NY, USA, 2001. ACM Press.

[27] Lawrence Livermore National Laboratory. IBM POWER systems overview. http://www.llnl.gov/computing/tutorials/ibm_sp/, July 2007.

[28] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[29] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, New York, NY, USA, 2005. ACM Press.

[30] C. Lattner and V. S. Adve. Transparent pointer compression for linked data structures. In *MSP '05: Proceedings of the 2005 workshop on Memory system performance*, pages 24–35, New York, NY, USA, 2005. ACM Press.

[31] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 278–289, New York, NY, USA, 2007. ACM Press.

[32] Chris Lattner and Vikram Adve. Data structure analysis: A fast and scalable context-sensitive heap analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[33] J. Laudon. Performance per watt: The new server focus. *SIGARCH Comput. Archit. News*, 33(4):5–13, 2005.

[34] O. Lhoták, S. Curial, and J. N. Amaral. An optimal encoding to represent a single set in an robdd. Submitted as a brief contribution to IEEE Transactions on Computers.

[35] O. Lhoták, S. Curial, and J. N. Amaral. Using XBDDs and ZBDDs in points-to analysis. Submitted to Software Practice and Experience.

[36] O. Lhoták, S. Curial, and J. N. Amaral. Using ZBDDs in points-to analysis. In *20th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Urbana, Illinois, October 2007.

[37] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Compiler Construction, 12th International Conference*, LNCS, pages 153–169, Warsaw, Poland, April 2003. Springer.

[38] O. Lhoták and L. Hendren. Jedd: A BDD-based relational extension of Java. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 158 – 169, 2004.

[39] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction, 15th International Conference*, LNCS, pages 47–64, Vienna, March 2006. Springer.

[40] C. K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ACM SIGOPS Operating Systems Review*, pages 222 – 233, 1996.

[41] International Business Machines. Special effects. The IBM System Scientific Computing User Group: http://www.spscicomp.org/ScicomP12/Presentations/IBM/Tutorial_7.Special_Effects2.pdf.

[42] International Business Machines. Hardware overview. The IBM System Scientific Computing User Group: http://www.spscicomp.org/ScicomP12/Presentations/IBM/Tutorial_2.Hardware_AIX_Overview.pdf, July 2006.

[43] Nicolette McFadden. POWER5 architecture. http://www-941.ibm.com/collaboration/wiki/display/WikiPtype/POWER5+Architecture, February 2006.

[44] Patrick Meredith, Balpreet Pankaj, Swarup Sahoo, Chris Lattner, and Vikram Adve. How successful is data structure analysis in isolating and analyzing linked data structures? Tech. Report UIUCDCS-R-2005-2658, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Nov 2005.

[45] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC '93: Proceedings of the 30th international conference on Design automation*, pages 272–277, 1993.

[46] A. Mishchenko. An introduction to zero-suppressed binary decision diagrams. Technical report, Portland State University, June 2001.

[47] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 19 April 1965.

[48] S. S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[49] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[50] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *POPL*, pages 275 – 307, Portland, OR, USA, January 2002.

[51] R. M. Rabbah and K. V. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions on Embedded Computing Systems*, 2(2):1–32, May 2003.

[52] J. Rao and K. A. Ross. Making B+ - Trees cache conscious in main memory. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 475–486, New York, NY, USA, 2000. ACM Press.

[53] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. Program. Lang. Syst.*, 17(2):233–263, 1995.

[54] S. Rubin, R. Bodik, and T. Chilimbi. An efficient profile-analysis framework for data-layout optimizations. In *POPL*, pages 140 – 153, Portland, OR, USA, January 2002.

[55] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Compiler Construction*, pages 126–137, Warsaw, Poland, April 2003.

[56] M. Sauerhoff and I. Wegener. On the complexity of minimizing the OBDD size for incompletely specified functions. *IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15, 1996.

[57] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[58] K. Shin, J. Kim, S. Kim, and H. Han. Restructuring field layouts for embedded memory systems. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 937–942, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[59] R. Sinha and J. Zobel. Cache-conscious sorting of large sets of strings with dynamic tries. *J. Exp. Algorithmics*, 9:1.5, 2004.

[60] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

[61] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[62] A. Stoutchinin, J. N. Amaral, G. R. Gao, J. C. Dehnert, S. Jain, and A. Douillet. Speculative prefetching of induction pointers. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 289–303, London, UK, 2001. Springer-Verlag.

[63] D. N. Truong, F. Bodin, and A. Seznec. Improving cache behavior of dynamically allocated data structures. In *Seventh International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 322–329, 1998.

[64] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Compiler Construction, 9th International Conference (CC 2000)*, pages 18–34, 2000.

[65] D. Vianney, A. Mericas, B. Maron, T. Chen, S. Kunkel, and B. Olszewski. Cpi analysis on power5, part 1: Tools for measuring performance. http://www-128.ibm.com/developerworks/library/pa-cpipower1/, April 2006.

[66] J. Whaley. Joeq: a virtual machine and compiler infrastructure. In *IVME '03: Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 58–66, 2003.

[67] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog and binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 97 – 118. Springer-Verlag, November 2005.

[68] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, 2004.

[69] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[70] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral. *Forma*: A framework for safe automatic array reshaping. *ACM Transactions on Programming Languages and Systems*, to appear.

[71] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 255–266, New York, NY, USA, 2004. ACM Press.

[72] J. Zhu. Symbolic pointer analysis. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 150–157. ACM Press, 2002.

[73] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 145–157, 2004.

[74] Craig B. Zilles. Benchmark health considered harmful. *SIGARCH Comput. Archit. News*, 29(3):4–5, 2001.