Logical object tagging architecture
Owens, Howard Dewey
ProQuest Dissertations and Theses; 1997; ProQuest Dissertations & Theses Global

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Copyright

by

Howard Dewey Owens

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Logical Object Tagging Architecture

Approved by
Dissertation Comittee:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Logical Object Tagging Architecture

by

Howard Dewey Owens, B.S, M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1997

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

UMI Number: 9822681

UMI Microform 9822681
Copyright 1998, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

In memory of Herman Dewey Owens

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Acknowledgments

First I must thank Dr. Baxter F. Womack for his support and advise since the beginning of my work at The University. Next I would like to thank Dr. Joseph Rahmeh for serving as co-advisor and helping so much to improve this research. I would also like to single out Dr. Mario J. Gonzalez for his support and advise along the way and for serving as the chairman of my qualifying committee and co-author of my paper. Finally, I would like to thank Dr.s Joydeep Ghosh, Vijay Garg, and Martin Wong for serving on the dissertation committee and assisting to improve the results.

I must also thank several other people for assistance towards this dissertation. First I thank Quentin Barnes of Motorola. Although we have never met, Quentin has provided consistent and generous support by answering all manner of questions and supplying software for my research. Michael Cruess allowed me time and resources at Motorola to complete this dissertation. Jean-Paul Carcenac, also of Motorola, provided replacement hardware when mine failed.Without his generous loan my research could not have been completed. Jon Gettinger, of Pure Software, provided long loans of Purify before funding could be found to purchase the product. Tomas Evensen, of Diab Data, provided a C++ compiler. Dr. Susan Barber, of The University of Texas, read an early draft of the paper presented in Chapter 2 and provided helpful comments.

Most importantly, I must thank family and especially my parents. What they could not teach me in my formal education, they more then made up for by teaching, and setting a good example of, honesty, integrity, perseverance, and patience. Without these qualities, along with a healthy dose of common sense and the will to succeed, it would have been impossible to come this far.

Finally, I would like to thank H. G. Wells for his great stories which have helped keep me sane during these trying times and to director Erle C. Kenton for Lota, the panther woman, in *Island of Lost Souls*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Logical Object Tagging Architecture

Publication No.____________

Howard Dewey Owens, Ph.D.

The University of Texas at Austin, 1997

Supervisors: Baxter F. Womack and

Joseph Rahmeh

The Logical Object Tagging Architecture is presented. The major contribution of this architecture is features which allow compilers to control access to data in object-oriented software systems with minimum performance degradation. Compilers can limit data access to the objects that own them while supporting fine grained (small) objects.

Unlike previous systems which support objects by tagging pointers to objects with access rights information, this new architecture tags the data with object ownership information. In effect, this new architecture does not limit the address an object may generate (address space management) but limits the memory locations which it can access (storage space management). In addition, previous object-oriented systems provided a complete system secutity model requiring new operating systems with object support from the ground up including support for individual protection domains for each object. This new architecture provides the ability to enhance a traditional process based system to allow compilers to manage protection domains of objects within a process.

Enforcing object boundaries in object-oriented software systems can eliminate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

important types of defects across several classes of software defects. In addition, the tagging mechanism in this new architecture can be used to eliminate other important types of software defects by catching memory accesses which are known to be faulty.

The defect detection tool Purify® is used to provide a survey of software defects. This survey is used to motivate the new architecture and to provide a base of comparison. After the architecture is developed it is analyzed by using a cycle accurate simulator for the 88110 processor. This simulator is modified with instrumentation and features which allow penalties to be extracted for the new architecture when it is implemented with a base 88110 implementation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Table Of Contents

Acknowledgments v

Abstract vi

Table Of Contents viii

Chapter 1. Introduction 1

1.1. The Software Problem 2

1.2. Previous Attempts at Solution 3

1.3. Idea for a new architecture 4

1.4. What to Expect 5

Chapter 2. The Software Problem 6

2.1. Introduction 6

2.2. Software errors and Purify 9

2.2.1. Where software errors are found. 9

2.2.2. Software errors and memory access 10

2.2.3. Purify: making memory access observable. 10

2.2.4. Memory access error types 11

2.2.5. Interpreting output from Purify. 12

2.3. Experimentation 16

2.3.1. Methodology 16

2.3.2. Example session 17

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

2.3.3. Experiments 19
2.4. Results 24
2.4.1. Dynamic error data 25
2.4.2. Static error data 26
2.4.3. Analysis of data 27
2.5. Comparing C to C++ 28
2.6. Summary conclusions 31
Chapter 3. Previous Work 32
3.1. Multi-threaded Architecture 32
3.1.1. HEP 33
3.2. Capability-Based Systems 35
3.2.1. Plessey 250 37
3.2.2. IBM System/38 38
3.2.3. M-Machine Guarded Pointers 40
3.3. Object-based Systems 42
3.3.1. iAPX432 43
3.3.2. MUTABOR 45
3.3.3. Ra 47
3.4. Software Defect Detection Schemes 49
3.4.1. Purify 50
3.4.2. Safe-C 51

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 4. Logical Object Tagging Architecture 53

4.1. LOTA Reference 53

4.1.1. Overview 53

4.1.2. Programming Model 57

4.1.3. Addressing Modes and Instruction Set Summary 58

4.1.4. Instruction and Data Caches 60

4.1.5. Memory Organization 64

4.2. Programming Considerations 65

4.2.1. Domain Crossing 65

4.2.2. Stack Management 67

4.2.3. Heap Management 71

4.3. Qualitative Analysis 71

4.3.1. Defect Detection 71

4.3.2. Domain Crossing 74

4.3.3. Abstract Data Type Implementation 76

4.3.4. Resources 76

4.3.5. Stack Management 77

4.3.6. Heap 78

4.3.7. Simplicity 78

4.4. Segmented Alternative 79

4.4.1. 80386 Processor 80

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

4.4.2. Modified 80386 Segmented Architecture . . . 82
4.5. Information Flow . . . 86
4.5.1. Von Neumann Computer . . . 87
4.5.2. Tag Information Flow . . . 87
Chapter 5. Analysis . . . 89
5.1. Implementation of LOTA . . . 89
5.1.1. The 88110 . . . 89
5.1.2. 110L . . . 91
5.1.3. Cost Analysis . . . 100
5.2. Experiments . . . 101
5.2.1. The Simulator - XSim . . . 101
5.2.2. Measurements . . . 104
5.2.3. Software Tested . . . 116
5.2.4. Example Session . . . 118
5.3. Results and Analysis . . . 119
5.3.1. Tag Transfers . . . 119
5.3.2. Domain Crossing . . . 123
5.3.3. Stack Space . . . 125
5.3.4. Heap Space . . . 127
5.3.5. In-line Domain Crossing . . . 130
5.3.6. Summary . . . 134

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 6. Refinement 138
6.1. Tightening the Bounds 138
6.1.1. Cache. 138
6.1.2. Domains 140
6.2. Improving Performance 142
6.2.1. Tag Transfer 142
6.2.2. Stack Space 142
6.3. Improving Security Against Defects 147
6.3.1. Domain Entry Points. 148
6.3.2. Domain Return 148
6.4. System Issues 149
6.4.1. Disk Transfer 149
6.4.2. Tag Memory 150
6.5. Application of Tags 152
6.5.1. Uninitialized Read Tags 152
6.5.2. Boundary Condition 152
6.6. Alternative Applications for LOTA 153
6.6.1. Second Level Cache 153
6.6.2. Smart Card Memory Protection 157
Chapter 7. Summary and Conclusions 160
7.1. The Software Problem 160

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

7.2. Logical Object Tagging Architecture 161

7.3. Future Directions 162

7.4. Conclusion 163

Appendix A. Full Text of Electronic Correspondence 164

Appendix B. Additional Data for Purify Experiments 175

Appendix C. XSim Experimental Data 182

Bibliography 242

VITA 247

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 1

Introduction

A recent discussion in the Internet news group *comp.arch* was on the subject of array index checking. It was generally agreed that out-of-bounds array accesses are a problem and a problem which should never occur in released software systems. The method to eradicate this problem was a point of heated debate. The perceived problem was blamed on hardware, the user, and everything in-between. Each participant in the discussion had his/her own perception of where the problem is and therefore where a solution should be applied.

One potential solution is to use memory segments for each array. Memory segments have their start and end addresses clearly represented by registers in the processor. It was noted that this solution caused extra pressure on the register resources of the processor and that a bounds check must be made for each array access [A.1 on page 164]. The compiler might be able to optimize some of these checks away, but many expressions used for array index values are much too complicated for a correctness proof within the compiler [A.1].

Another potential solution for C programs is to use a preprocessor to convert the source code into a safe version of the program such as in the system Safe-C [1]. Although it was noted that C made safe is very inefficient [A.2], it was still used as an argument against putting special checking features in hardware in the event that such systems become efficient over time making the extra hardware obsolete [A.3].

A common and reoccurring theme in the discussion was the possibility of a feature, either hardware or software, that could be turned on during program debugging and then turned off for final release of the software. The benefit of this method is that inefficiencies are only tolerated during program development while the released software receives the benefit of more reliable software without paying the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

penalty of less efficient software. It was noted however, that software testing is never complete and the possibility of defects always exists [A.4,A.5]. Chapter 2 demonstrates the problem of software defects, including out-of-bounds array accesses, escaping the testing process and finding their way into released software systems.

1.1 The Software Problem

Out-of-bounds array indexing is only one piece of the defect detection and correction part of the software problem. In the larger picture, defect detection is part of the software life cycle. The "Software Problem" is that the software life cycle is too expensive, too difficult to predict, and too hard to manage. When the software life cycle is divided into the popular categories of analysis, design, implementation, testing, and maintenance it is evident that defect detection and correction covers all phases of the software life cycle. There are tools which attempt to detect defects while a program is running, but these tools have design constraints which make their operation too inefficient to be used by the customer.

One obvious way to eliminate the part of the problem which is defect detection and correction is to eliminate the source of the defects in the first place. There are computer languages, such as **forth** and **pascal**, which have checks against many defects either at compile time or with run time software. But it would be impractical to require all programmers everywhere to use a given language system to declare the defect detection and correction part of the software problem solved. First, solutions cannot apply to part of the problem without considering the entire software life cycle. Second, it is market forces which tend to promote a computer language. These market forces have promoted C to a prominent position. C is a high level programming language which not only increases the efficiency of implementing software (the solution to a problem), but compiled code is also very run time efficient. Part of its run time efficiency is derived from pointer manipulation and the lack of automatic array bounds checking.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Object-oriented analysis and design is a methodology to improve the respective parts of the software life cycle, often referred to as the object-oriented software life cycle. The C++ programming language allows the object-oriented analysis and design to be carried over into implementation. The major contribution to the popularity of C++ is that it maintains the familiarity of syntax with C and much of C's efficiency while at the same time offering features to allow object-oriented programming. The object-oriented life cycle is a unifying paradigm which allows objects to be identified in the problem domain during analysis, solution objects to be identified during design, and the possibility of additional objects to be applied during implementation of the solution. In fact, it would be optimal if the objects of each phase could be expressed in the implementation language [2].

For this research two facts are important, object-oriented programming has become popular, and C++ is the dominant object-oriented programming language. Given these facts this research is concerned with the structure of object-oriented software and the effectiveness of C++ as an implementation language with respect to the software life cycle. In particular, the division of data and the routines that manipulate that data lends itself to protection domains between objects. Such protection between objects can eliminate many software defects and bound many others.

1.2 Previous Attempts at Solution

There have been several attempts at building more reliable software systems. Capability-based systems divide programs into procedures and data segments and limit a procedure's ability to address anything outside of its own data segments. Object-oriented systems combine the segmented data and capability features with the natural structure of objects. Capability-based and object-oriented systems can be designed in software to use traditional computer hardware or they can be designed to be supported directly in hardware. A common theme of these systems is the attempt to control the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

program addresses generated by a procedure or object within a program to access data. These program addresses are called logical or virtual addresses because they are typically translated by the hardware and operating system into physical storage addresses.

This research will present a survey of several of the important systems which have been presented over time in the area of capability-based addressing and object-oriented hardware architecture. In addition to these important systems the survey will cover other systems designed to aid in the detection of software defects at run time.

1.3 Idea for a new architecture

It is observed that object-oriented software does present favorable organization of data in software systems and that it does make sense to provide protection between objects at run time in order to contain important types of software defects. Such systems have not been successful in commercial terms and it is often shown that inefficient implementations are the reason for such systems failing to gain popularity. Time-shared operating systems have long had protection between programs sharing the system at any given moment. The protection mechanisms commonly used in time-shared systems are too inefficient to apply to protection within a program at the object level.

It is observed that in every case reported to date protection implies limiting the addresses a program is allowed to generate. These are program addresses, or virtual addresses, and as such must be checked before the address is translated by hardware. Checking program addresses before translation is referred to as address space management. This type of address management takes place within the processor. Since processor implementation is limited by the resources available (transistors) it is difficult to justify increased resources.

The question for this research is: can a mechanism be offered to processes within traditional computer systems which allows protection between objects of a given

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

program while being efficient enough to use during the entire software life cycle? It is proposed here to tag physical memory with object ownership. The basic idea is to move the resources required to perform protection to the storage side of the memory address translation hardware. Physical memory is tagged with the identification of the object which owns it. This contrasts with previous systems which tag a pointer (i.e. the logical address) with permissions for accessing the corresponding memory. The proposed tags can be transferred into the cache of the processor as part of normal data movement. A data access can then be checked for ownership by the currently executing object within the cache. Since the check is in the cache, which is already a tagged mechanism, it is hoped that checking for object ownership can proceed at cache speeds. This research will explore the implementation of such an architecture and analyze the result.

1.4 What to Expect

This research explores the software problem, including the array indexing problem, in C and C++ software systems. Chapter 2 presents previous work in exposing the problem associated with defects in released software systems; it also presents the results and analysis for an original survey of such defects. Chapter 3 presents a survey of previous software and hardware solutions to many of the software problems and includes several of the more interesting alternatives. Chapter 4 develops the architecture for this research while Chapter 5 analyzes a possible implementation. Chapter 6 will revisit the architecture in an attempt to refine it in light of the analysis.

It is claimed that this new architecture will provide for the efficient run time protection of objects within a traditional process. Such protection will bound the defects in software to object boundaries. In addition, other important software defects can be detected by the tagging mechanism. The architecture is efficient enough to be used during the entire software life cycle.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 2

The Software Problem

Perfect quality represents 100% conformance to specifications. Complex systems make it difficult and expensive to assure conformance to specifications by post production testing alone. As a result quality assurance processes are moving upstream in the life cycle, i.e., statistical process control and design for manufacturability. These processes try to avoid defects or make it easier to detect defects. In software systems this move to upstream processes for quality control is still in its early stages. As with more traditional manufacturing systems, maturity of the software design and development process will dramatically reduce the cost of attaining conformance to specifications. Even so, it is difficult to imagine a "bug free" complex software system. Downstream processes will continue to play an important part in the efforts to achieve defect-free software. This chapter presents results of a survey which used the defect detection tool Purify to examine off-the-shelf software products in order to show that software errors continue to escape testing and threaten field failures [3]. Errors detected are compiled and presented. All data were collected on C and C++ programs running in a UNIX operating system environment. The information contained in this chapter was presented in [3] but is expanded here and contains a few minor corrections[1].

2.1 Introduction

The software life cycle can be divided into two parts: development and maintenance. Development has been broadly divided into analysis and design, implementation, and test processes [4,5]. Most efforts in software engineering have

---

1. © 1996 IEEE. Reprinted, with permission, from Proceedings of the International Conference on Software Maintenance, Monterrey, CA, November, 1996, pp 104-113.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

been directed toward these development processes. This is not surprising. Borrowing from experience with more traditional manufacturing systems, it is accepted that defect prevention saves money as compared to defect correction as a means to achieve conformance to specifications [6].

In order to assess the savings of defect prevention over a software product life cycle, data are needed for the cost of attaining quality through defect prevention. The cost to fix defects is the cost of not attaining quality. For this cost data are available [7]. However, for the cost of attaining quality through defect prevention, there is very little data. The lack of tracking metrics in the software development process is a major contributor to this deficiency. There have been efforts to model the cost of software quality based on the Capability Maturity Model [8]. However, it is estimated that ninety percent of software organizations today are at the capability level of software development described as ad hoc, undefined and chaotic [8]. Organizations at this low capability level experience a large percentage of cost during the life cycle associated with correcting software defects.

In some ways the software life cycle is not similar to product life cycles of traditional manufacturing systems. While a car may wear out, a program will not. For a car, maintenance is required to keep it operating within design specifications. Maintenance of software is quite different. It is estimated that sixty to seventy percent of the entire software life cycle is devoted to maintenance [4,9]. However, upon closer examination, software maintenance is found to be divided into changes to meet user requirements, enhancement changes, and bug fixes [4]. The inevitable part of software maintenance is evolutionary changes. Fixing bugs is the only part of maintenance which has the potential to be completely eliminated.

The primary quality factor to the user is conformance to specifications [9]. As the requirements change other quality factors become apparent. These include robustness and extendibility. Internal quality factors such as modularity and readability become very important to meet the quality factors visible to the user. The goal of these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

internal qualities is to make possible the user visible qualities in a timely fashion in the face of evolving specifications. The combined goal of user visible and internal qualities is to reduce the overall costs associated with the software life cycle by reducing maintenance costs [10]. Software defects are the most ominous trait visible to the user.

This chapter presents the results of research using a tool, Purify®, to discover symptoms of software defects, namely memory access errors, in released software systems. These memory access errors are classified and compiled. Although memory access errors do not include all types of logic errors, they do include most memory overlay errors which were previously reported as the most difficult to analyze and correct and which had the greatest impact on system availability [11]. In addition, memory access errors include those errors which are least likely to be caught during testing. This is true probably because testing primarily focuses on conformity to specification. Even though a thorough specification may include code coverage requirements for testing and expected behavior in the face of out-of-bounds inputs, a defect usually must manifest itself as a deviation from specification to be caught in testing. Memory access errors are an important type of defect which continue to escape testing yet maintain a strong potential for manifesting themselves as deviations from the specification.

All data were collected on Sun SPARC workstations running the UNIX operating system version SunOS 4.1.3. This was the first combination of hardware and operating system software on which Purify was available. The programs examined are all C and C++ programs. Only memory access errors from each software package and their respective libraries are reported. Memory access errors from operating system library code were intentionally excluded to avoid attributing defects to the delivered software package.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

2.2 Software errors and Purify

2.2.1 Where software errors are found

Software errors are found in all phases of the software life cycle. Of course, the earlier they are found the less costly they are to fix. When a software error is found during one of the development processes it is often called an *internal error* because the error was found internal to the organization developing the product.

Once a product has been released it becomes more costly to correct errors. The cost comes from finding a fix, testing the fix, releasing the new version and updating any affected documentation. When a software error is found after release it is often called an external error because the error was visible external to the organization developing the product.

Sullivan and Chillarege performed excellent studies of the IBM RETAIN (REmote Technical Assistance Information Network) database of field failures and corrective actions [11,12]. In their studies several error types were grouped together and classified as *overlay* errors. Overlay errors are defined as any error which results in memory corruption. Typical error types classified as overlay errors are:

- Allocation Management: A module releases a memory region before it has finished using it. This error is also referred to as the dangling pointer error.
- Copying Overrun: A memory copy operation overruns the destination buffer.
- Pointer Management: A pointer variable is corrupted causing unrelated memory to be over-written.

Just these three types of errors accounted for nineteen to twenty seven percent of all errors sampled for three products [12].

Surveys such as the one done by Sullivan and Chillarege used pre-existing databases created by field service divisions to track problems, fixes, and release information. The data reported here are different in the fact that they were actively sought out as opposed to extracted from a pre-existing database even though both types

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

of databases are collections of external errors.

2.2.2 Software errors and memory access

One of the biggest problems associated with locating a software error is finding a symptom to observe. Consider the simple example $c = a + b$ where the variable a was unintentionally left uninitialized. If this statement is part of a large module it may take much time and effort to find and fix if only the module input and output values are observable.

Now consider the same statement, $c = a + b$ with variable a uninitialized, in terms of memory access. The error is that variable a is used before it is initialized. In terms of memory access the memory location holding variable a is read (used) before it is written (initialized). A read before write memory access is defined as an uninitialized memory read error.

If memory accesses could be made visible it would open up the opportunity to capture the use of uninitialized variables. It would also make possible the capture of many of the overlay errors of the type allocation management, copying overrun, and pointer management identified earlier.

2.2.3 Purify: making memory access observable

Purify is a software tool which assists a programmer in locating software errors by making memory accesses observable. Purify does this by replacing, at link time, all memory access instructions with a jump to one of its own subroutines to check the memory access, record profiling information, and perform the access. In addition, Purify intercepts calls to the UNIX subroutines *malloc* and *free* to keep track of memory allocated to a program. Building a software application with Purify results in a new version which has additional error checking code and is therefore referred to as the *instrumented* version.

Purify allocates memory to hold two state bits for each byte of program memory. Each byte of program memory can therefore be in one of four states and each state

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

defines the type of memory access considered valid. The three states used by Purify and their valid memory accesses are defined as follows:

1. *unallocated*: The corresponding byte has not been allocated and any access is in error.

2. *allocated*: The corresponding byte has been allocated but not initialized. A read access is an error while a write access is valid. A write access will change the state to *initialized*.

3. *initialized*: The corresponding byte has been allocated and initialized. All access are valid.

The experiments for this work included use of Purify versions 1.1, 2.0 and 2.1. The latter versions tended to make error reports more usable in ways such as batch reporting of errors at the end of a run with redundant information removed. Programs instrumented with Purify consumed approximately five times their normal CPU time. Starting with Purify version 3.0 the tool has stressed an interactive graphical interface which will make surveys such as this work more difficult. More detailed information on the technical details of Purify is found in [13] and more information on the product Purify can be found in the user's guide [14].

2.2.4 Memory access error types

As Purify monitors memory accesses and calls to *malloc* and *free* it updates the state bits accordingly. When an access is not considered valid Purify reports a memory access error. Five types of memory access errors were compiled for this work. These are described in the following sections.

2.2.4.1 Uninitialized memory read error (umr). A program variable was read before it was initialized. This error was found to occur with the most frequency. This is partially explained by how UNIX C and C++ programmers expect uninitialized global data to be filled with zeros.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

2.2.4.2 Array bounds read error (abr). A memory location in the unallocated state and adjacent to an array was read. Purify surrounds arrays with buffer memory in the unallocated state in order to catch array accesses which go out of bounds.

2.2.4.3 Array bounds write error (abw) . A memory location in the unallocated state and adjacent to an array was written. Programs which are not instrumented with Purify may behave differently than their instrumented counterparts because *abw* errors will overwrite adjacent data. The buffer memory around arrays in instrumented versions provide a measure of protection from overwriting. Without such tools as Purify these types of errors are often very difficult to find.

2.2.4.4 Free memory read error (fmr). An unallocated memory location was read. This programming error was observed in this research to be common. One observed example was in the management of a linked list of data structures. In a linked list each element contains a pointer to the next element in the list. When elements of such a list are de-allocated the programmer sometimes de-allocates an element before reading the link to the next element.

2.2.4.5 Free memory write error (fmw). An unallocated memory location was written.

2.2.5 Interpreting output from Purify

2.2.5.1 Example output. To illustrate the error categories introduced earlier, consider the program in Example 1. Line numbers have been added for convenience.

Example 1.

```
1  main(){
2      int a,b,c;
3      a=1;
4      c=a+b;
5  }
```

Figure 1 shows the output from Purify. Purify reports an uninitialized memory

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

```
Purify (umr): uninitialized memory
   read:
  * This is occurring while in:
   main          [line 4, main.c,
   pc=0x1b338]
   start         [crt0.o, pc=0x2064]
  * Reading 4 bytes from 0xf7ffeea8 on
   the stack.
  * This is local variable "b" in
   function main.
```

Figure 1 Purify output for simple umr error.

read error, *umr*, on line 4 as emphasized in boldface. The error reported is a *umr* error due to the fact that the memory area used to store *b* was read before it was written.

2.2.5.2 Differentiating system errors. Application software is made up of its own source code plus any library routines supplied by the operating system. Consider the program in Example 2.

Example 2.

```
1  #include <stdio.h>
2  main(){
3     char c[10];
4     printf("%s",c);
5  }
```

This trivial program calls the operating system supplied library function *printf.* The output from Purify is shown in Figure 2. Notice that Purify reports an uninitialized memory read error as indicated in the boldface line. Also, the line number of the offending source statement is missing. Library routines are most likely to have minimum symbolic information available. Purify does what it can by reporting the program module and program counter.

The fact that Purify omits line numbers was relied on to identify which errors were from system supplied libraries and which were from application source code. Errors from the system routines were excluded when compiling the results of this

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

```
Purify (umr): uninitialized memory
   read:
  * This is occurring while in:
   _doprnt        [doprnt.o,
   pc=0xf753829c]
   printf         [printf.o,
   pc=0xf7546678]
   main           [line 4, example2.c,
   pc=0x2655c]
   start          [crt0.o, pc=0x2064]
  * Reading 1 byte from 0xf7ffeea4 on
   the stack.
  * This is local variable "c" in
   function main.
```

Figure 2 Purify report of system library.

work.

2.2.5.3 Call stack. Example 3 contains a program which performs the same calculation as in Example 1 except that it contains a `main` function and two levels of subroutines.

Example 3.

```
1  main(){
2     void fun1(int*,int*);
3     int a,b;
4     a=1;
5     fun1(&a,&b);
6  }
7  void fun1(int* a, int* b){
8     void fun2(int*,int*);
9     fun2(a,b);
10 }
11 void fun2(int* a, int* b){
12    int c;
13    c=*a+*b;
14 }
```

The *main* routine calls *fun1* which in turn calls *fun2* to perform the statement $c = a + b$. The error is clearly in *main* where the variable *b* is not initialized. The symptom of the error is the uninitialized memory read error while reading the variable *b* in *fun2*. Figure 3 shows the report from Purify for this error. The lines in boldface

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

```
Purify (umr): uninitialized memory
   read:
  * This is occurring while in:
   fun2         [line 13, example3.c,
   pc=0x1b438]
   fun1         [line 9, example3.c,
   pc=0x1b3e4]
   main         [line 5, example3.c,
   pc=0x1b384]
   start        [crt0.o, pc=0x206c]
  * Reading 4 bytes from 0xf7ffeea8 on
   the stack.
  * This is 8 bytes below frame
   pointer in function main.
```

Figure 3 Purify call stack example.

show how Purify reports the subroutine calling sequence. From top to bottom it shows that *fun2* was called by *fun1* which in turn was called by *main*. This information may prove useful in determining the program error which leads to the memory access error. In this case it would lead the programmer to the function *main* where the variable *b* was not initialized.

Now consider the program in Example 4 which differs from Example 3 only slightly.

Example 4.

```
1  main(){
2     void fun1();
3     fun1();
4  }
5  void fun1(){
6     void fun2(int*,int*);
7     int a,b;
8     a=1;
9     fun2(&a,&b);
10 }
11 void fun2(int* a, int* b){
12    int c;
13    c=*a+*b;
14 }
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

In this example *fun1* declares *a* and *b* but only initializes *a*. This is the root of the error. However, the symptom of the error, like in Example 3, is the uninitialized memory read in *fun2*.The report from Purify is shown in Figure 4.

```
Purify (umr): uninitialized memory
   read:
  * This is occurring while in:
   fun2          [line 13, example4.c,
   pc=0x1b408]
   fun1          [line 9, example4.c,
   pc=0x1b3b4]
   main          [line 3, example4.c,
   pc=0x1b36c]
   start         [crt0.o, pc=0x206c]
 * Reading 4 bytes from 0xf7ffee48 on
   the stack.
  * This is local variable "b" in
   function fun1.
```

Figure 4 Purify call stack variation.

The programming errors in Examples 3 and 4 are the same: variable *b* is left uninitialized. However, in Example 3 it is *main* and in Example 4 it is *fun1*, which make this error. As such they are two different programming errors. The symptom of these two programming errors which Purify detects is the memory access error in *fun2*. In this body of work for a given program the calling sequence is not considered while totaling the access errors.

2.3 Experimentation

2.3.1 Methodology

Purify supports a limited number of hardware platforms, operating systems, and programming languages. The supported combination of platform and operating system most readily available for this work was Sun SPARC workstations running the SunOS 4.1.3 version of UNIX. The languages supported are C and C++ which are of particular interest for this work due to their popularity.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

A total of fifteen software packages were tested as shown in Table 2 in Appendix B on page 175. These packages contain thirty one independent programs of which sixteen are C++ programs, fourteen are C programs, and one program is of mixed C and C++. Source code was acquired for each of these packages so that Purify could instrument the code. To assist Purify in identifying line numbers of memory access errors, each package was compiled with compiler options to maximize symbolic information.

Instrumented programs were tested by running them under conditions to which they should be able to exactly perform their tasks according to their specifications. No abnormal conditions were used to stress an instrumented program under test.

2.3.2 Example session

This section presents three example program fragments which contain memory access errors. These program fragments are excerpts from software used for initial experience with Purify for this work.

2.3.2.1 Free memory read error. Consider the program fragment in Example 5.

Example 5.

```
for(p=first; p != nil; p=p->next) {
    free(p);
}
```

In this example program excerpt, a linked list of elements is freed returning the storage space back to the free memory storage allocator. On each pass through the loop an element of the list is freed and then the next element pointer is retrieved. Purify reports a free memory read error on each pass through the loop in this example. The reason for this is that the next element pointer is retrieved from the current element after it has already been freed.

The designer of this software did not consider this an error. The reason given is that calling the UNIX library subroutine *free* only returns the storage to the free

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

memory pool. The memory itself is not made inaccessible nor are the values stored in the memory corrupted. The memory access to retrieve the pointer to the next item in the list, therefore, seems safe. However, this is a dangerous programming practice which may adversely affect future maintenance requirements. Even within the UNIX environment, more UNIX implementations are becoming multi-threaded while hardware is more commonly multiprocessor [15]. These evolutionary changes may invalidate these implementation dependent assumptions about free store.

2.3.2.2 Array bounds write error. Consider the code fragment in Example 6.

Example 6.

```
for(i=0; i<MAX; i++) {
   via[i] = BLOCKED;
}
```

In this code excerpt a loop is used to initialize each element in an array with the constant *BLOCKED*. Unfortunately, the array is dynamically sized with either two or three elements. *MAX* is a constant defined to be three. Therefore, when a run of the software dynamically allocates a two element array for the problem, the above code fragment will write off the end of the array by one location. Purify reports an array bounds write error. It was determined that the constant *MAX* was left over from the prototype implementation of the software and remained only due to an oversight.

2.3.2.3 Array bounds read error. Consider the code fragment in Example 7.

Example 7.

```
for(i=0; i<MAX; i++) {
   b = c[i];
   d = c[i+1];
   ...
}
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

In this code excerpt an array bounds is overstepped by one element in the statement $d = c[i + 1]$ on the last iteration through the loop. The code later tests for this case and allows for it. The resulting calculation is correct and most UNIX programmers assume that reading one value past the end of an array is safe. This is true when the byte past the array is still accessible to the program but may be a dangerous assumption for future maintenance requirements. Purify reported an array bounds read error.

2.3.3 Experiments

The following ten sections describe the experiments for the fifteen software packages. Six of the GNU packages were part of the same experiment and are reported together as GNU tools.

2.3.3.1 Custom Cell Synthesizer. The Custom Cell Synthesizer (CCS) is an Microelectronic and Computer Technology Corporation (MCC) Computer Aided Design tool set. It takes as input a sized transistor network listing for a cell. The tools produce a symbolic physical layout for a CMOS circuit. The CCS tool set is a research prototype and the project developing this tool set is no longer active. The final version of CCS, CCS Version 2.5, was used for this research. (Refer to the CCS design specification for more information about CCS [16].)

CCS comes in two parts, the proprietary CCS tool set and the required public domain support programs and libraries. Both parts were installed according to the installation manual [17]. The source code was then instrumented with Purify.

The CCS release comes with an example workshop directory, and instructions on how to run examples are described in the user's guide [18]. After a few simple setup steps the commands shown in Example 8 were used to test CCS.

Example 8.

```
make
make CCS/adder_4row.log
make CCS/adder_4row.xyclog
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

The result of running these commands is to invoke a set of tools in the CCS tool suite to compute a cell layout from its netlist. The tools invoked include the instrumented versions of the programs *GateToGate*, *XCellCompact*, *XCellPlacer*, *XCellRouter*, *placement*, and *router*.

2.3.3.2 GNU tools. The Free Software Foundation's GNU tools are an excellent source of software for this type of research. It is a common practice to retrieve electronically GNU software packages from one of the repositories of GNU software. Each of these software packages is released as a single composite file which has been compressed to save space. Each time a software package is retrieved several steps must be followed to compile and install the package. These steps are repeated for each package. In general these steps include:

- Running *zcat* to uncompress the file.
- Using *tar* to break the composite file into its components.
- Running *make* to compile and install the package.
- *make* performs calls to several other utilities to perform its work including the compiler, *sed*, and *awk*.

For this research the GNU software packages *gzip*, *tar*, *make*, *gcc*, *sed* and *awk* were compiled and instrumented with Purify. The GNU debugger *gdb* was then imported and compiled using these instrumented versions of tools. The compiler used was an instrumented version of the GNU compiler consisting of the compiler driver *gcc*, the code generator *cc1* and the compiler preprocessor *cpp*.

During the build of *gdb* several of the instrumented GNU tools were executed multiple times as demonstrated in Table 1.The result was to generate, edit, compile, and link the individual source code files into the program *gdb*.

Program	**Runs**
make	14

Table 1: Execution Runs

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Program	Runs
gcc	143
cc1	141
cpp	141
awk	2
sed	264

Table 1: Execution Runs (Continued)

2.3.3.3 groff. Another GNU software package is *groff*, the *nroff/troff* compatible text formatter. The *groff* package includes *geqn*, *gpic*, *grodvi*, *groff*, *gsoelim*, *gtbl*, and *gtroff*. A paper written for *troff* was used to test the package [19]. The command line used to test *groff* is given in Example 9.

Example 9.

```
groff -etps -Tdvi -me paper.input
```

2.3.3.4 InterViews. InterViews is a software system for building graphical interfaces for window based applications [20]. InterViews is object-oriented with windows, buttons, menus, and documents as active elements with inherited behavior. A programmer uses InterViews as a set of class libraries to aid in building the interactive view of data for the application.

For this research four programs built with InterViews were tested. These programs included *doc*, *ibuild*, *idraw*, and *iclass*. The program *doc* is a simple *what you see is what you get* (WYSIWYG) document editor. It was tested by opening and viewing several of the example documents provided with the InterViews release.

The program *ibuild* is an interactive interface builder. It allows a programmer to interactively edit the interface of an application. The programmer graphically creates objects from the InterViews library and interfaces the code generated by *ibuild* with application code. *ibuild* was tested by working through the *ibuild* user's manual [21].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

As part of this process, the program *idraw* was spawned. *idraw* is a drawing editor. *ibuild* has a palette of tools which have built-in graphical representations. However, *ibuild* will call *idraw* to allow the user to build a more general graphics image. *idraw* was tested in this fashion.

The program *iclass* is an interactive class browser. It allows the user to efficiently browse the class hierarchy in an application. *iclass* was tested by browsing various C++ source code segments.

2.3.3.5 ghostscript. Ghostscript is a GNU package which allows the viewing of Postscript documents in an X Windows environment. For testing, the document "Solaris Application Level Multithreading Seminar: Participant's Guide" [22] was opened and paged through.

2.3.3.6 emacs. During the compilation of *emacs* an intermediate program called *temacs* is built. This program is then used to build the standard *emacs* program with the command shown in Example 10.

Example 10.

```
temacs -batch -l loadup.el dump
```

For this research the command *temacs* was instrumented with Purify and data collected as *temacs* built *emacs*. No memory access errors were found in the supplied code. However, looking at the memory access errors within system library functions, Purify reported 64 uninitialized memory read errors, 872 array bounds read errors, and 3,720 free memory read errors. All were within the *write* function call, part of the standard C library of input/output functions. In addition, all were reported with the same program counter value within *write* and each time *write* was called from the routine *sys_write*, a function inside *emacs* source code.

2.3.3.7 ldl++. *ldl++*, a *L*ogical *D*ata *L*anguage, is a deductive database system [23] designed for use in knowledge-based applications [24] requiring efficient access to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

large collections of data. The instrumented version of *ldl++* was tested using the commands as shown in Example 11. The results of this test were curious due to the complete absence of the common array bounds read errors and uninitialized memory read errors but the high count of free memory read errors and free memory write errors.

Example 11.
```
ldl++(1)> open bike
ldl++(2)> initdb bike.fac
ldl++(3)> compile
ldl++(4)> query partsof(bike, B)
ldl++(5)> query partsof(bike, [spoke,
   rim, sprockets, bolt, nut, tube,
   casing, rearderailleur, rearbrake,
   chain, chainrings, frontderailleur,
   pedals, saddle, post, fork,
   frontbrake, stem, bar, cables,
   levers, grips, headset])
ldl++(6)> exit
```

2.3.3.8 amd. *amd* is the file system automounter for Berkeley Software Distribution version of UNIX, 4.4 BSD [25]. The function of *amd* is to automatically mount filesystems which are in use and to unmount these filesystems when they are no longer in use.

Since *amd* intercepts all operations on remote filesystems, it must run as a daemon process at root privilege level. The instrumented version of *amd* was allowed to operate while several remote filesystems were accessed, after which the daemon was shut down and the error data collected. During normal operation *amd* forks child processes to perform tasks; 54 such processes were forked during this test.

2.3.3.9 parprosys. *PARallel PROduction SYStems* (Parprosys) is a parallel architecture for serializable production systems [26]. The software includes a simulator for the architecture as well as a compiler for Parallel Production Language, PPL, an explicitly parallel production language [27].

This software package represents the only pre-release package considered for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

this research. To test the system the command shown in Example 12 was executed.

Example 12.

```
pplc -N 4 mab.1
```

Several errors were reported but the single cause of most of these errors was due to a syntax error in the source code. The C++ source code statement in question is shown in Example 13.

Example 13.

```
heap = new  event_ptr(array_length+1);
```

The intent of this code to create an array of things. However, the parentheses indicates that the parameters inside are constructor arguments, not a count of array elements. The GNU compiler did not complain about this statement. The correct statement should have contained brackets instead of parentheses.

The code as tested caused a single element array to be created where an array of *array_length + 1* was expected. For this reason many array bounds read and write errors were reported by Purify.

Its interesting to note that although C++ introduced many type safe improvements over C, this error would not have occurred in C.

2.3.3.10 sendmail. *sendmail* is the UNIX Internetwork mail routing facility from BSD [28]. Like *amd*, *sendmail* was run as a daemon process with root privilege. It was tested by replacing the normally running *sendmail* with its instrumented counterpart which was allowed to run for one working day. During this day 427 child processes were spawned to perform normal service.

2.4 Results

This section presents data collected for this work. Data are presented in two ways. The first, in bar chart form, presents frequency of error types. The second, in pie

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

chart form, provides statistics on percent of total for each error type. Section 2.4.1 defines and presents dynamic error data. Section 2.4.2 defines and presents static error data. Analysis of the data is saved for Section 2.4.3.

2.4.1 Dynamic error data

Dynamic error data is defined as the sum of occurrences for memory access error types. Since this research involves memory access errors, which are symptoms of programming errors, each access error represents an opportunity for a symptom to be propagated to other parts of the system.

Dynamic read errors have the potential to propagate errors by being included in calculations of intermediate values. When used in pointer arithmetic these read errors could cause additional access errors as well.

Dynamic write errors have the potential to propagate errors by corrupting memory locations used later by the same or even different parts of the system. The potential to propagate errors to unrelated parts of the system makes write errors the most troublesome and often the most difficult to isolate.

Figure 5 displays frequency of errors. There were a total of 25,576 access errors

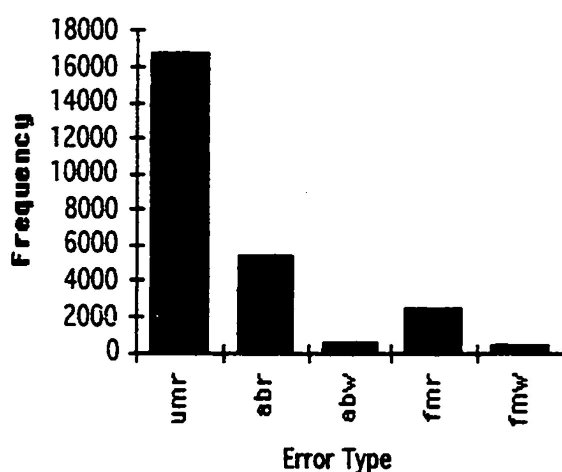


Figure 5 Dynamic error frequency

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

collected for this effort. Figure 6 shows the statistics for dynamic errors.

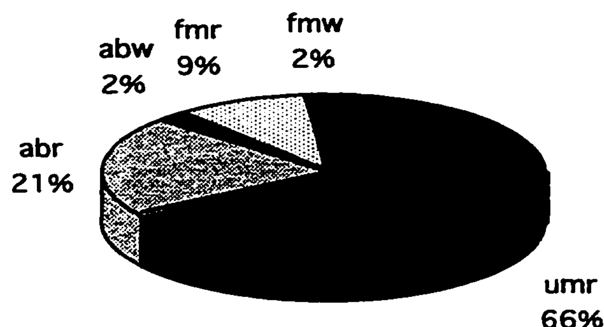


Figure 6 Dynamic statistics

2.4.2 Static error data

Static error data is defined as the number of source lines responsible for memory access errors. This represents the number of source lines involved in generating symptoms. The static error frequency gives an idea of how often an error type is coded.

Static error frequency is shown in Figure 7. There were a total of 209 different application source lines involved in generating access errors. Statistics on static error frequency are given in Figure 8.

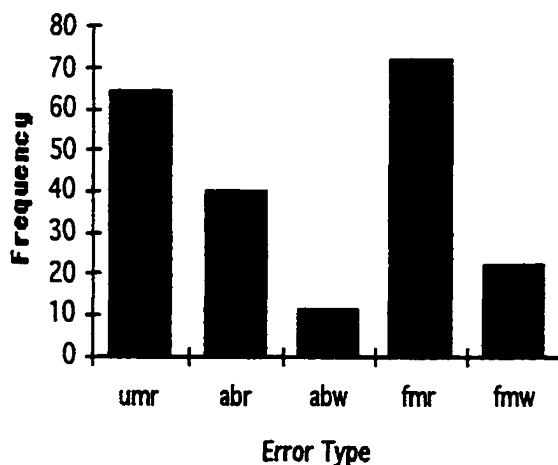


Figure 7 Static error frequency

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

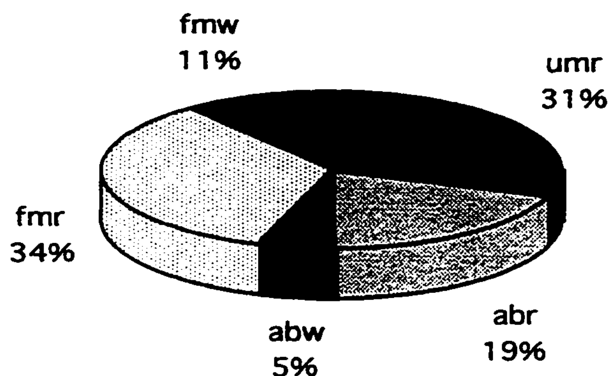


Figure 8 Static Statistics

2.4.3 Analysis of data

2.4.3.1 Uninitialized memory read errors. In the dynamic error data uninitialized memory read errors occurred with the most frequency. A sampling of the source code for this error type points to sloppy coding practice as the cause. In some cases an uninitialized pointer or variable was passed to a subroutine knowing it would not be used based on other arguments in that same call. In other cases an uninitialized permanent variable is compared to another variable apparently expecting its incidental value to cause the desired test results.

Comparing the dynamic and static occurrences of uninitialized memory read errors show that this error type accounts for a much larger percentage of dynamic errors than it does for static errors. This suggests this error type is much more likely to be found in loops.

2.4.3.2 Array bounds errors. The dynamic occurrences of array bounds read errors occurred with the second highest frequency. Many of these errors appear to be due to mismanagement of dynamically allocated arrays designed to handle strings. These character arrays are commonly scanned until a special terminating character is found or some maximum length is reached. As long as these errors are confined to read type

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

errors, program operation is usually not affected. However, array bounds write errors are more serious. In one case a program had 317 dynamic array bounds write and read errors which could all be accounted for by a single statement. The loop stepped a location beyond the end of two equally sized arrays and assigned one to the other.

2.4.3.3 Free memory errors. Free memory read and write errors together accounted for only eleven percent of the total dynamic errors. However, for static errors, this group accounted for forty five percent of the total. This suggests that free memory read and write access errors are much more likely to be scattered around the source code. It could be argued that free memory read errors are benign in many cases where the free memory is accessed immediately after being freed. However, there is no corresponding argument to support free memory write errors.

2.5 Comparing C to C++

Up to this point data from C and C++ programs have been combined. However, it is interesting to compare the two programming languages to see if anything can be learned about their relative error rates for access errors. Parprosys was excluded since it is made up of both C and C++ code. Figure 9 shows the dynamic frequency for each error type for the C and C++ programs considered. Figure 10 shows the static frequency for each error type. Care must be taken when comparing the data in these figures. To make a direct comparison a figure of merit for the programs must be given. Intuitively, there may be more lines of code in one or the other language. Less intuitive is the expressive power of a language which may affect the number of source code statements required to implement a software solution. Nevertheless, it can be seen that in raw numbers more uninitialized memory read, array bounds read, and array bounds write error types occurred in the C programs, and more free memory read and free memory write error types occurred in the C++ programs.

To get a more direct comparison, consider Figures 11 and 12 which normalize the numbers to percentages with respect to each language. It is interesting to note that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

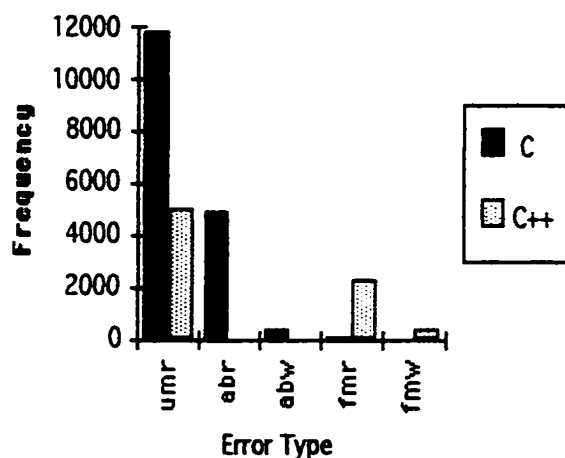


Figure 9 Dynamic frequency comparison

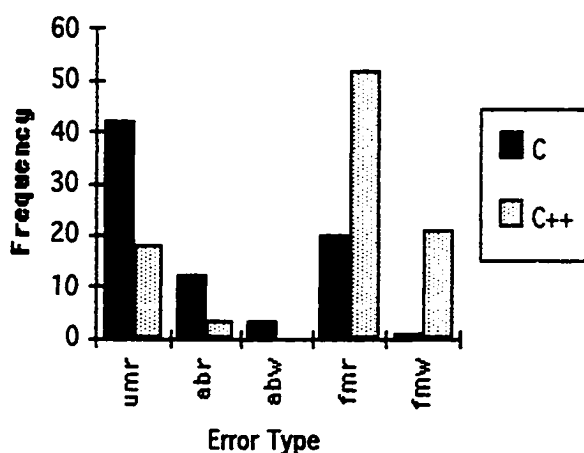


Figure 10 Static Frequency Comparison

the dynamic behavior of the C and C++ programs with regard to uninitialized memory read errors is similar (69% for C versus 65% for C++) but the static behavior is quite different (54% for C versus 19% for C++). Its also interesting to note that static values of free memory read and free memory write error types were larger than for their

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

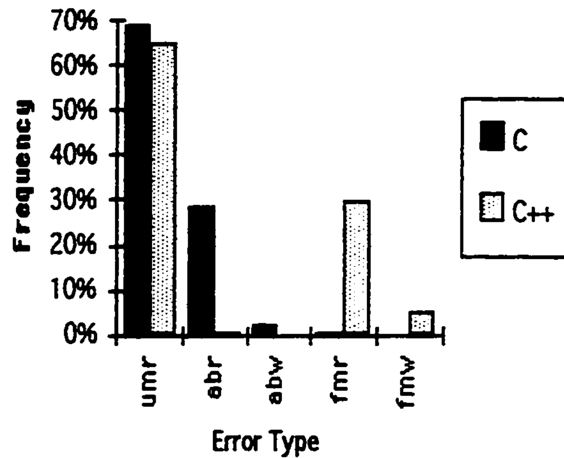


Figure 11 Dynamic statistical comparison

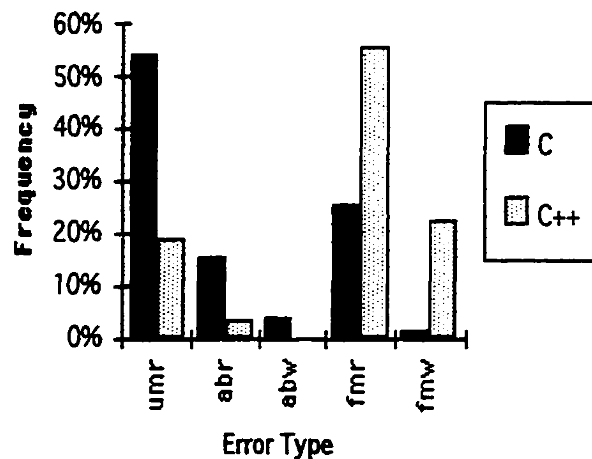


Figure 12 Static statistical comparison.

dynamic counterparts in both languages. This could point out the difficulty with user (explicit) memory management in both C and C++. It also points to free memory access errors being much more likely to be scattered around the application source code in both languages.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

2.6 Summary conclusions

In this work Purify was used to actively search for memory access errors in software systems and prove that software errors continue to escape testing and threaten field failures. Data on five types of access errors were collected. These errors consisted of uninitialized memory read, array bounds read, array bounds write, free memory read and free memory write errors. Data on 25,576 access errors representing 209 source code statements were collected. The data were collected from fifteen software packages consisting of thirty one independent programs. Sixteen C++, fourteen C, and one C/C++ programs were tested. Each program under test was observed during normal operation and no attempt was made to stress the program. Even so, the volume of data suggest there is plenty of room for improved processes to eliminate software defects.

The data show that read type errors occur with more frequency than write type errors. Many of these errors are considered benign and have no effect on the output produced, which is probably why they make it through the testing process. Write errors tend to be more catastrophic, and the smaller frequency of occurrence of this type of error was expected. The data also show that free memory access errors account for a larger percentage of all errors in C++ programs, a result not anticipated.

Many of the errors reported in this work could be flagged by improved compilers. In particular, compilers could issue warnings when variables are used before they have been initialized. Some compilers already do this. Improved cooperation between compilers, operating systems, and hardware could also help eliminate many of these errors.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 3

Previous Work

There have been many important systems in the past which have attempted to address some of the concerns of reliable software systems. Some of these systems have included complete new hardware architecture and operating system software. Others are software only methods to detect defects within a program. This chapter presents important work in four areas: multi-threaded computer architecture, capability-based addressing, object-oriented systems, and software defect detection systems.

3.1 Multi-threaded Architecture

In a multi-threaded architecture several threads of program execution have active state within the processor at any given time. Resources such as the program counter and working registers are duplicated in such a way that instructions may execute from any thread ready to run. The processor is designed to switch between threads to execute their instructions. Instructions in one thread are independent from instructions of another thread so there are no resource conflicts. The idea is to maintain maximum processor utilization by splitting a program into many independent threads. This idea is extended to multiple processors and many programs (possibly cooperating programs).

The multi-threaded architecture is a solution to a different problem than this research is primarily concerned with. Multi-threaded architectures attempt to solve the massively parallel super computer utilization problem. However, the mechanisms used to support multiple threads sharing the processor in a secure fashion is very interesting to consider for its protection properties. In such architectures threads which are part of a program (task) share a single protection domain. Threads from different tasks may be active simultaneously. The pioneering work for this type of architecture was the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Denelcor Heterogeneous Element Processor (HEP) multi-computer [29].

3.1.1 HEP

In the HEP system terminology, a program is assigned a protection domain as part of a task [30]. A task is a set of one or more processes corresponding to independent threads of execution. A task begins with a single process and expands to as many processes as it needs up to a pre-declared maximum for the task. The machine can start one instruction per cycle from any given process of any task, but instructions cannot be issued from a given process more often than once every eight cycles (pipeline depth). Each clock cycle the machine switches to a new process, and this includes a protection domain switch if necessary.

The system can support 64 active user processes at a time and 8 active user tasks at a time (likewise for operating system processes and tasks). To support this many active execution states at one time the processor has 2048 general purpose registers. A protection domain for a task is identified by a task status work (TSW), a 64 bit quantity which can be held in a single machine register. The components of the TSW are shown visually in Figure 13. Registers, program memory and data memory are all identified by base and limit values in the TSW and the hardware enforces register and memory usage within these limits for the corresponding task. The remaining base value is an index into an array of constants. Each process of a task has a process status word

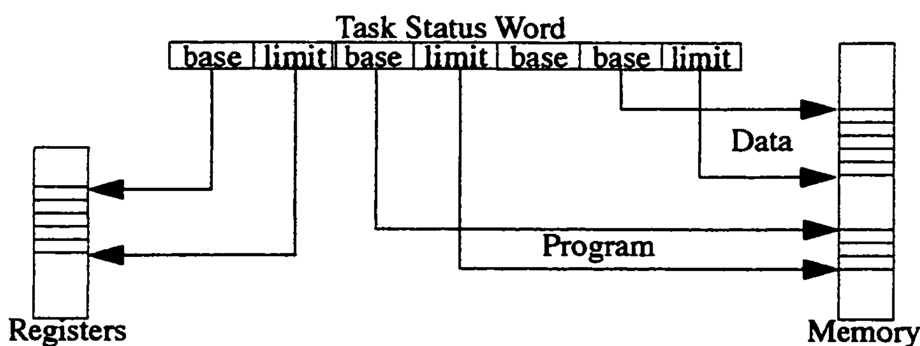


Figure 13 HEP Task Status Word

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

(PSW) which contains a 20 bit program counter and some state information.

Processes of a HEP task share a common protection domain which is completely specified by its corresponding TSW. The speed at which the HEP can switch between protection domains make it worth considering for application to object-based protection domains. The HEP is able to switch protection domains so quickly (single cycle) because it can completely specify the protection domain (TSW) in a processor register. However, it only contains eight such registers in the processor for user TSWs. To protect one object from another each object would have to be a task level code segment. This would mean only eight objects of a single program could fill the machine instead of the designed eight tasks (programs).

To be an effective architecture for object-based protection HEP would have to support many more protection domains. One possibility would be to expand the process status word (PSW) to include protection within a task. There are enough general purpose registers to support the processing requirements of 64 such objects. The PSW would have to be extended with many of the same fields as the TSW, but these fields could perhaps be smaller since they are specifying sub-protection within a normal task protection domain. This possible architecture change overlooks the important feature of HEP. Processes are independent threads of code ready to have instructions issued. The ordinary object-oriented program is considered a sequential program with a single thread of execution passing through it. As such, only one object is active at any given instance of the program execution. In HEP, a process does not exist if it does not have a program counter from which instructions are ready, or waiting, to be issued. Objects would therefore not have their protection domain loaded into the processor until it has been called. HEP provides no feature to support the fast loading of new protection domains. In addition, a task in HEP must declare up front how many processes it will require in order to schedule the resources needed.

Considering the HEP as a possible platform for object-based protection is an interesting exercise. On one hand it points out a possible method to provide for quick

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

protection domain changes as would be required by lots of small objects. But to make use of the mechanism in HEP the state of an object would have to be present in the processor. The HEP limit of eight user protection domains is much too small for even a single object-oriented program. On the other hand, expanding HEP to object-based protection shows how the resource requirements in the processor would have to grow to support lots of objects with individual protection.

Since this research is presenting a new type of tagged computer architecture, it is interesting to note that HEP is also a tagged computer architecture. The most common use of tags in computers is to identify the type of a data item, such as an integer or floating point value. HEP, as in the proposed architecture for this research, uses tags in a unique way. In HEP memory includes an extra bit for indicating if the corresponding word of memory is empty or full. When used, the word cannot be written when its already full, and it cannot be read when it is already empty. This mechanism is used for program synchronization, not support for protection, and will not be discussed farther.

3.2 Capability-Based Systems

Capability-based systems are segmented memory architectures which require a capability to be presented before access to a memory segment is allowed [31]. Capabilities themselves act as tickets presented to the hardware to perform certain accesses on a segment. A capability uniquely identifies a single memory segment and the access rights the owner of the capability possesses. In addition, capability-based systems make it impossible for a capability to be forged or the access rights to be modified. In this way the architecture can assure a given program module will remain within its own allocated memory segments. Cooperating modules may copy the capabilities to promote sharing of memory segments.

In capability-based addressing all references to memory are made indirectly through a capability as shown in Figure 14. This figure shows a possible virtual address

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

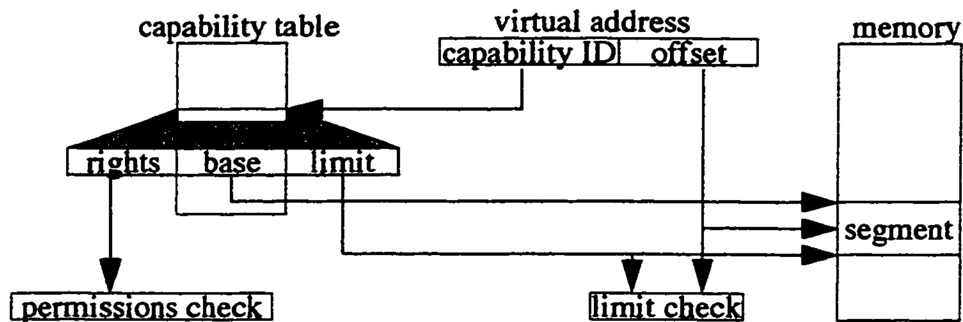


Figure 14 Capability-based Addressing

format where part of the address specifies a capability.The capability contains the base address of the memory segment and its size.The base address is combined (added) with the offset contained in the virtual address to find the data. Also, the limit in the capability is compared to the offset to check for out-of-bounds references.

The need for privilege modes of operation (i.e. operating system versus user program) are eliminated by the use of access rights which cannot be forged or modified. Typical access rights include read, write, and execute. A more complex right is the *enter* right. For one program module to call another module it must have a capability with the enter right. The call instruction will create a new protection domain for this new module to execute. In this way program modules are protected from each other. The capabilities of the new module are a combination of those given to it when it was created and those passed to it through parameters.

Capability-based systems are intended to be used where protection domains are small and changed often. Typically a program module, such as a subroutine, will be contained in its own segment. In this way the protection domain is tailored during a program execution to suit the needs of its sub-modules.

The advantage of small and dynamic protection domains is the facilitated software debugging and the confinement of errors during maintenance. There are two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

basic methods used to implement capabilities. The first maintains capabilities in separate segments and is manipulated with special machine instructions. The second tags capabilities in bits invisible to programs and allows data and capabilities to be freely mixed in data segments. The following two sub-sections present a commercial implementation of each of these methods. Section 3.2.1 presents the Plessey 250 which exemplifies segregating capabilities and data. Section 3.2.2 presents the IBM System/38 which exemplifies the mixing of data and capabilities in normal data segments. In Section 3.2.3 the guarded pointers of the M-Machine are presented as a more modern and technologically advanced example of capability-based addressing.

3.2.1 Plessey 250

The Plessey System 250 computer has two types of instructions [31]. The store mode instructions reference memory (store). Direct instructions perform operations which do not require a memory operand. Eight capability registers (C0-C7) identify the protection domain at any instance during the execution of a process. Instructions themselves are referenced using the capability in register C7. Store mode instructions operate as shown in Figure 15. The instruction specifies a capability register which the hardware uses to infer the rights to access the segment and limit the offset value. Simple instructions allow capabilities to be transferred between capability segments and capability registers. To load a capability, as Figure 15 indicates, a capability identifies the segment where the rights are extracted as well as an index. The operating system maintains a system capability table in which the index identifies an entry to extract the base/limit pair the capability is to have. In this way different capabilities can have different rights to a given segment.

The computer requires no privileged mode of operation. The same facilities for protection domain management are available to the operating system and user programs alike. The difficulty with this architecture lies with the hierarchy of protection domains. It is entirely up to the user, or the compiler used, to manage the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

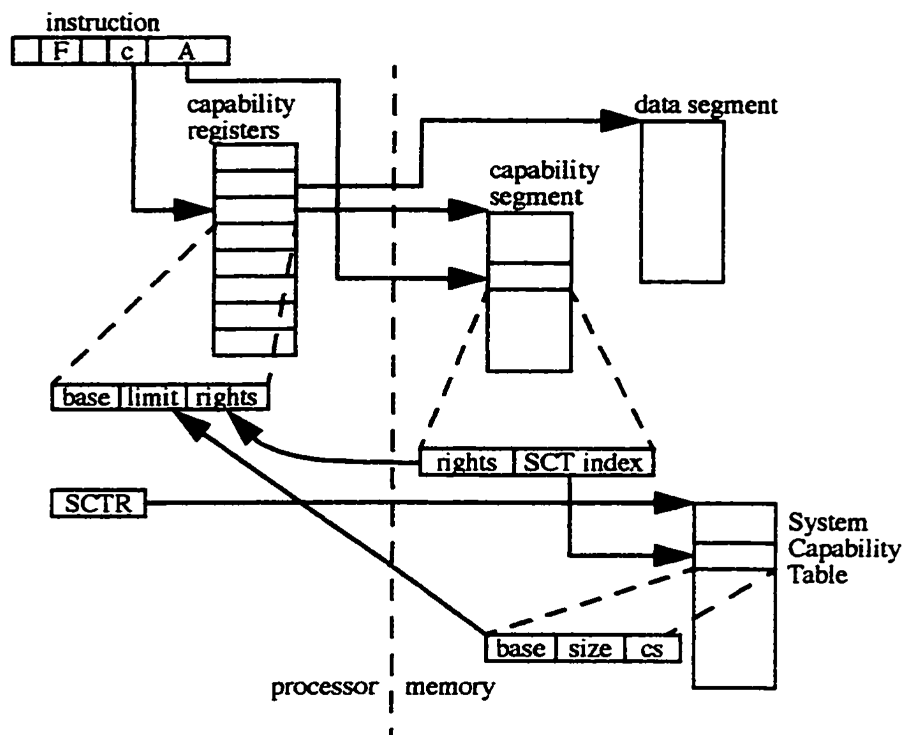


Figure 15 Addressing in the Plessey 250

capability registers. Capabilities are inherited through the capability registers and the hierarchy of capability segments which they point to. The architecture does not allow subroutine calls within a segment, therefore each subroutine must occupy its own segment. Capabilities being maintained in separate capability segments cause the proliferation of small segments. Also inherent in this architecture, as in most capability-based addressing architectures, is the indirection of memory access.

3.2.2 IBM System/38

The IBM System/38 is also a capability-based machine [31,32,33]. Its most unique feature is its use of tags to maintain capability integrity. This allows capabilities and data to be freely mixed in data segments. In System/38 a capability is called a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

pointer and is 128 bits long. Memory is 32 bits wide which means pointers take up four consecutive locations. Each word in memory contains an additional seven bits of error correction codes and one bit to indicate the location is part of a valid pointer. These eight bits are not part of the visible memory word and represent a 25 percent overhead in memory requirements.

Figure 16 shows the process of object addressing in System/38. Memory is maintained as segments and addresses are 64 bits. Instructions reference a pointer to gain the rights to access another segment. Microcode must extract the pointer and check that all four valid bits are set indicating a valid pointer. The rights field must then be checked for an allowed operation. Hardware directly supports only 48 bits of the 64 bit address so microcode must emulate the upper sixteen bits by checking them against a copy maintained in the segment header.

Although capability-based addressing in the System/38 is maintained on a segment basis, virtual memory is based on paging. The System/38 attempts to have the

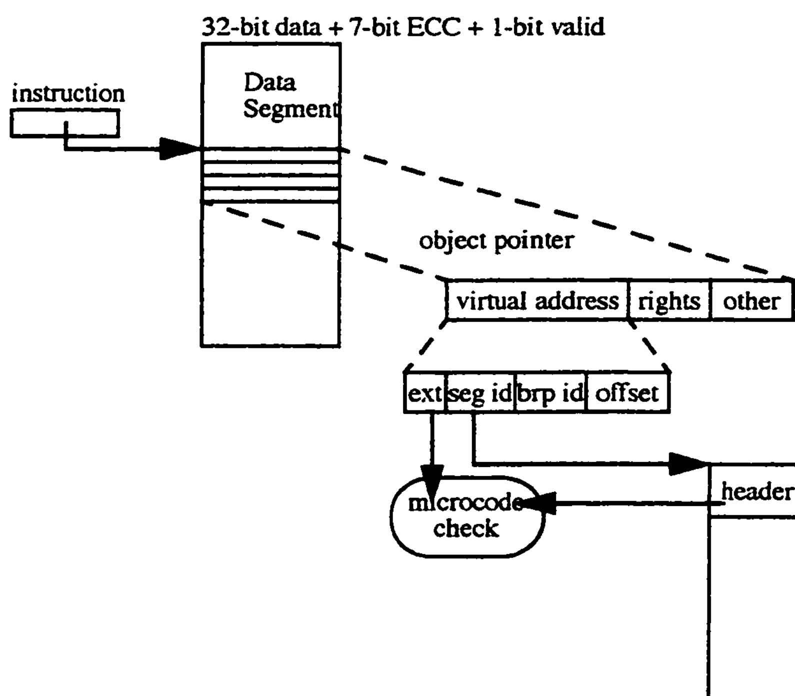


Figure 16 Object addressing in System/38

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

advantage of structuring programs into natural sized segments for protection domains while keeping the efficiency of a paged virtual memory system.

System/38 is a very complex architecture. The machine is implemented with two levels of microcode. Microcode must extract 128 bit pointers from 32 bit memory, validate the pointer and the access rights, translate the virtual address in the pointer to a physical address of the object, retrieve the upper 16 bits from the segment header and compare to the upper 16 bits in the pointer before access is allowed to the object. Although several steps of this process could possibly be optimized through technological enhancements, the inherent two levels of indirection to access the object would still remain.

3.2.3 M-Machine Guarded Pointers

One of the more interesting implementations of capability-based addressing is the guarded pointers of the M-Machine [34]. The fact that pointers are tagged to assure authenticity is very similar to the System/38 method to assure capabilities cannot be modified or forged yet can be freely mixed with data in a memory segment. But where the guarded pointer differs from its System/38 predecessor is in the fact that the capability is completely encoded into the pointer eliminating the need to look up the capability in a system wide capability table. This eliminates one level of indirection inherent in previous capability-based addressing architectures.

Figure 17 illustrates the guarded pointer format. The pointer is 64 bits in length

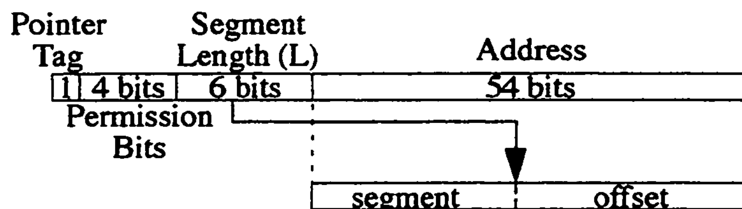


Figure 17 Format of a guarded pointer.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

and contains 54 bits for a virtual address. Processes in the M-Machine share a single 54 bit virtual address space. The permissions bits (rights) indicate what permissions the current process has to access memory. All memory accesses must be performed using guarded pointers. Any attempt to access memory otherwise will raise an exception. The ability for a guarded pointer to completely encode a capability lies in the segment length field (L). A segment in the M-Machine is simply a virtual address with 2^L least significant bits set to zero. The offset is a virtual address with the segment bits set to zero. In other words, L determines where to split a virtual address between its segment part and its offset within the segment part. Obviously this method to encode a segment location and size has some severe constraints, but allows a very efficient representation of a capability. Assuming completely random object sizes, such an allocation scheme would expect to incur a 25 percent waste in memory utilization.

The application of guarded pointers in the M-Machine solves the tough problem of allowing multiple threads of execution, from separate protection domains, to securely share processor resources in a multi-threaded computer architecture. Since all processes share a 54 bit virtual address space, and guarded pointers protect memory segments within this space, a process maintains security while the processor interleaves instructions from processes on a cycle-by-cycle basis.

The M-Machine provides for the rapid change of protection domains between multiple independent threads of execution. However, installing a new protection domain (i.e. when one object calls another) is still a rather heavy weight operation when compared to calling an object in the typical non-protected implementation. An object-based system implemented with guarded pointers would require a new protection domain to be loaded on each object call. Since guarded pointers are freely transferred between memory and registers (protected in both places with a tag) the calling routine would have to be careful to clear all registers of pointers before executing the enter capability instruction (enter right to a program segment). In addition, the calling routine would have to set up a return capability and pass it as an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

argument so the called routine would have permissions to return execution. The called routine must take similar precautions to keep from returning permissions to access its segments. As long as arguments to the called routine fit into registers there is no additional overhead, but when arguments exceed the capacity of registers a shared segment must be set up to place the arguments and a capability to access this segment passed to the called routine. However, to create a new segment for holding arguments requires a new guarded pointer be manufactured to address it, an operation which requires a separate call to a privileged routine capable of setting the tag bit.

3.3 Object-based Systems

In the computer language area the principle of information hiding and abstract data types have evolved to solve the same problem as capability-based addressing, namely the facilitated software debugging and the confinement of errors during maintenance. An abstract data type is a model which encompasses a type and an associated set of operations [35]. Abstract data types utilize the information hiding principle and tightly bind operations and data so that a change in the data structure defining the type will have a very localized effect. Localizing the effect of change has been motivated by the difficulty in maintaining complex software systems. Object-based systems combine capability-based addressing for protection with enhancements to aid in the implementation of abstract data types.

The following sections will present three architectures which support object-based systems. Section 3.3.1 presents the iAPX432 processor and exemplifies the construction of an entire system from scratch, both hardware and operating system, to support objects. Section 3.3.2 describes the protection scheme used by MUTABOR, a system which attempts to provide a technological solution to the problems encountered in the iAPX432. Section 3.3.3 describes the protection mechanism in Ra, a system which avoids the problems associated with adapting capability-based addressing and attempts to use an enhanced memory management unit to support objects.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

3.3.1 iAPX432

The Intel iAPX432 evolved from the Carnegie-Mellon Hydra operating system [31,36,37]. Its architecture is very complex and is heavily microcoded. Everything in the iAPX432 is an object. Capabilities are called Access Descriptors (ADs) and are the only protection mechanism. The machine is a stack architecture and contains no registers.

Figure 18 shows how object addressing is performed in the iAPX432. Protection domains are maintained on a segment basis, just as in the Plessey 250 and System/38. However, a segment is divided into two parts, the access part (AP) and the data part (DP). As in the Plessey 250 capabilities are maintained separately from data. A fenced segment approach is used to allow a single segment to contain data and/or capabilities.

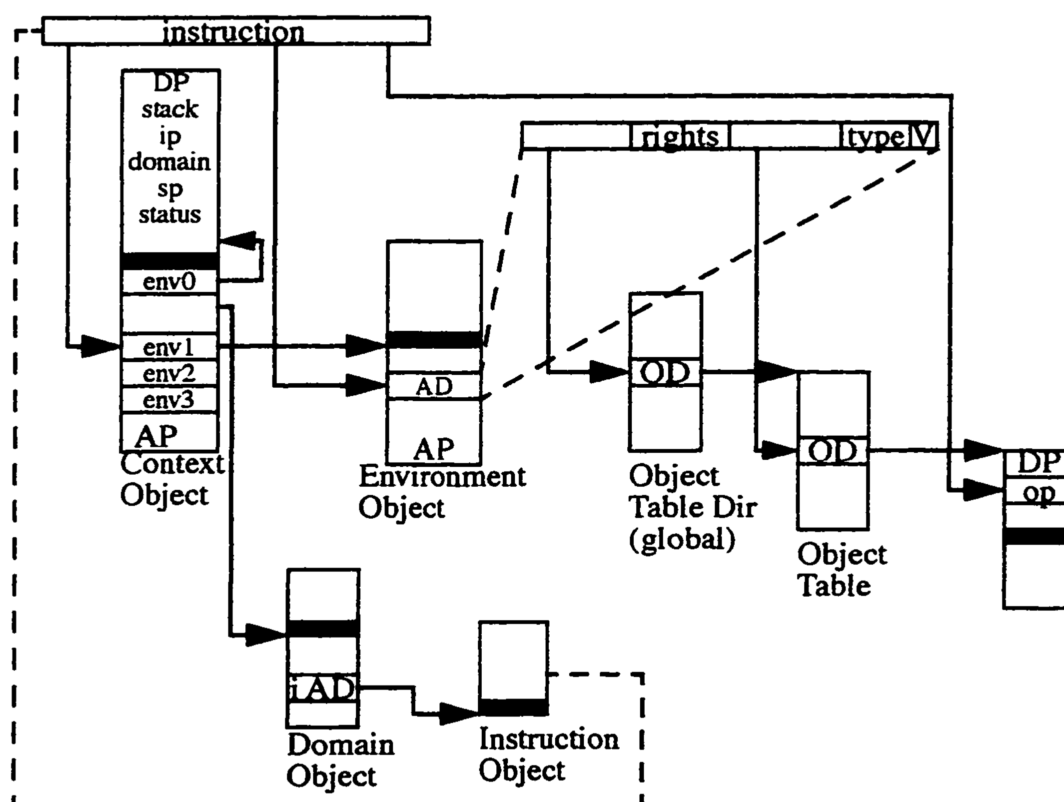


Figure 18 Addressing in the iAPX432

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Capabilities are kept on one side of the fence (AP) and are accessed only by microcode while data is maintained on the other side of the fence (DP). The Context Object maintains the context in which a procedure is operating. It contains the currently executing procedure's stack, instruction pointer and domain, stack pointer as well as the base set of capabilities the procedure has.

Object addressing in the iAPX432 is performed in two steps. For the first step the Context Object contains four special capabilities, *env0* through *env3* in Figure 18. These capabilities point to the current operating environment addressable by the procedure. These locations in the Context Object may actually be written by the procedure to change the current environment, in very much the same way capability registers in the Plessey 250 are updated. In the first step a field in the instruction selects which *env* capability to use. The one chosen selects an associated Environment Object. Another field in the instruction selects one of the capabilities (AD) in the Environment Object (offset into the AP side of the object). Now a capability has been selected for the object the instruction wishes to access. This capability contains the rights to access the target. Two offset fields in the capability select an object in the Object Table Directory and the Object Table, respectively. The Object Descriptor selected in this two level lookup operation now points to the target object. A final field in the instruction selects an offset into the data part (DP) of the target object. Exclusive of the instruction fetch, this two step procedure to address an object takes five memory accesses.

In the iAPX432 even instructions are kept as objects. The Context Object contains the current Domain Object AD (not shown in Figure 18) which points to the current Domain Object. The Instruction Domain AD (iAD in Figure 18) within the Domain Object points to the currently executing Instruction Object. The Context Object maintains an instruction pointer (ip) as an offset within the Instruction Object for instruction sequencing.

Although this description is very brief for such an architecture it points out the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

complexity involved. When the *call* instruction is executed the context must be changed. This is done by creating a new Context Object. Even though Context Objects are maintained in a free list of available Context Objects, initializing a new context is costly. Performance of the iAPX432 was not good. In fact, some point to it as an example of why architectures should be simplified. The iAPX432 not only suffers from the inherent two levels of indirection (capability lookup and physical location lookup), but adds considerable indirection and complexity in its implementation.

3.3.2 MUTABOR

MUTABOR is a object-oriented memory management co-processor to the Motorola 68020 processor [38,39,40]. The MUTABOR co-processor attempts to address the issues which caused the iAPX432 to have poor performance. The first of two key ideas is the separation of address space management and storage space management. The second is a large translation lookaside buffer to cache address translations. Like the iAPX432, MUTABOR supports a segmented memory with capability-based addressing to objects of the fenced segment type. Capabilities are maintained by microcode on the capability side of the fence.

The separation of address and storage management is revealed in the addressing scheme depicted in Figure 19. Virtual (program) addresses are issued by the 68020 and translated by MUTABOR. The virtual address is divided into three parts. The first part indexes into the root of a two-level tree and identifies a capability list by physical address. The second index picks a capability from this list. As in other capability-based systems, the capability identifies the segment type and rights to the segment. However, instead of a base and limit field the capability contains the virtual page number, referred to as the object short name, which contains the object.

The object short name is itself divided into three parts. The first two are used to index into a two-level page table to identify the frame in physical memory. Frames are considered large compared to objects and are therefore divided into sixteen parts.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

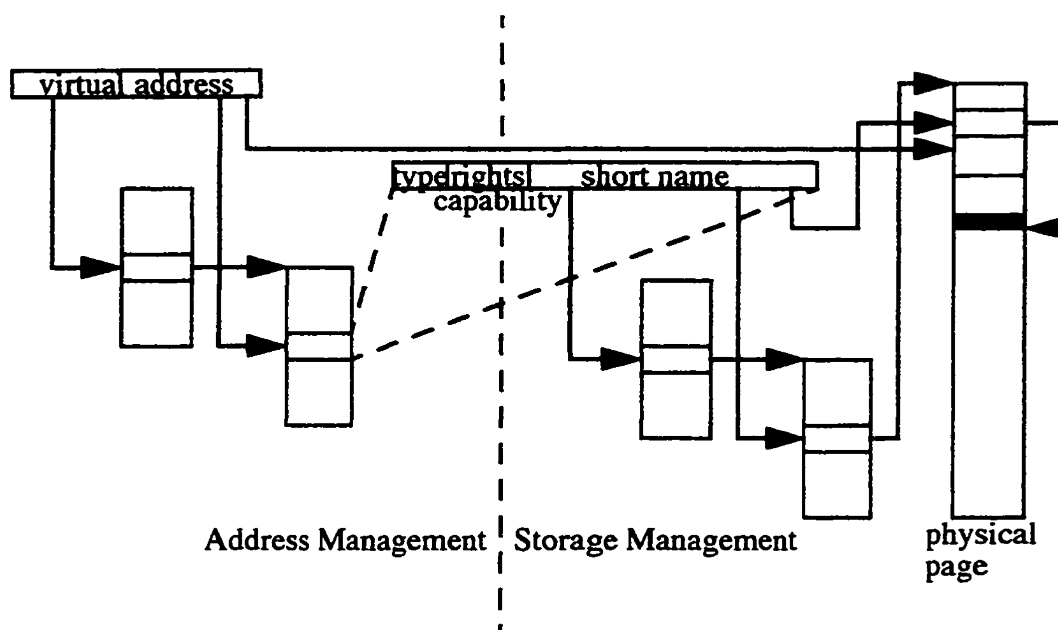


Figure 19 Addressing in MUTABOR

Sixteen words are reserved at the beginning of the frame as headers to identify the base and limit of an object within the page. The final part of the object short name chooses one of these headers. The final part of the virtual address is used as an offset from the base address in the header to address a memory word.

The addressing description above indicates five memory references must be made to perform address translation. A translation lookaside buffer is provided with separate object translation and page descriptor caches. Virtual addresses are extended by a four bit process identifier (PID) which separates the translation lookaside buffer into 16 sections reserved for 16 active processes. Each section is intended to be large in order to have a high hit rate.

The contribution of MUTABOR is its separation of management tasks so that each may have its own policies. The large translation lookaside buffer attempts to reduce the penalties of the inherent indirection in object-based systems. However, instead of reducing the amount of indirection as compared to the iAPX432, MUTABOR actually increases the levels of indirection. A very large and complex

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

caching scheme is required to accelerate the common case of memory access and very high hit rates are absolutely necessary. In addition the cache itself now requires two levels of lookup, the capability lookup and the physical address lookup, which must be performed in serial fashion. Compared to the iAPX432, MUTABOR liberally applies caching mechanisms to overcome the implementation limitations but otherwise brings no notable advances in accelerating capability-based addressing when used to implement object-based systems.

3.3.3 Ra

Ra is the kernel to the Clouds operating system, an object-based system. The proposed Ra architecture for object-based support is different from the other architectures surveyed in that it does not use capability-based addressing [41,42,43]. In Ra an object invocation is much like a context switch in traditional systems which provide protection on a process basis. Part of the overhead when making a switch in context is the manipulation of the memory management unit hardware as well as the performance hit from flushing and reloading the translation lookaside buffer.

The Ra memory management unit divides the virtual address space into four sections by using the two most significant bits of the address, similar to the R2000 processor [44]. The four spaces are kernel unmapped space, kernel mapped space, process space, and object space. The system loader is responsible for putting sections in the correct space. The Ra memory management unit takes the R2000 approach to translation lookaside buffer loading which is to leave it up to the kernel.

The three approaches to efficiency which the Ra memory management unit uses are:

- tagging virtual page numbers (VPNs) within the memory management unit to allow multiple protection domains to share the translation lookaside buffer,
- minimizing the memory management unit operations required to switch protection domains, and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

- use of a large translation lookaside buffer to minimize overflows.

The structure of the Ra memory management unit is partially shown in Figure 20. The upper two bits of a virtual address select one of the four spaces as mentioned

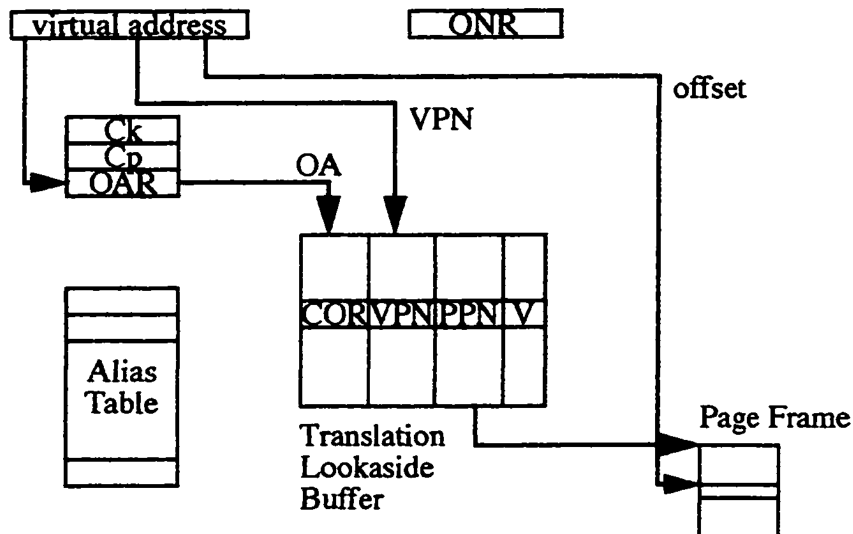


Figure 20 Proposed Ra Memory Management Unit

before. Object space addresses are tagged with the Object Alias Register (OAR) which contains an Object Alias (OA) for the currently executing object. The remaining bits of the virtual address are divided into VPN and offset. The OA and VPN are concatenated and used to do an associative search in the TLB to find the associated physical page number (PPN).

On object invocation the kernel must only write the new object name into the Object Name Register (ONR). The memory management unit maintains a list, called the object Alias Table (AT), of the most recently used object names. The index into the AT is used as the alias for the object in order to reduce the required bits in the translation lookaside buffer. The memory management unit must search the AT for an entry matching the object name. The index of the match is loaded into the OAR.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

The bottom line is that the Ra memory management unit extends the virtual address with a protection domain identifier to allow multiple domains to share the translation lookaside buffer. Additional features have been added to the basic idea to support a larger number of domains with acceptable implementation penalties. Adding domain identifiers to support multiple protection domains has been used many times but seldom on an object basis. Notice, however, the Ra memory management unit still assigns pages to a protection domain. In Ra, pages are recommended to be on the order of two kilobytes. This is much larger than the natural size of objects in most object-oriented programming languages.

The Ra memory management unit does not address other issues involved in switching tasks. In Ra an object invocation is like a task swap but with the advantage over tradition system that it requires a minimal amount of memory management unit overhead. It still requires a kernel trap to install the object alias of the new object to be executed and to copy any parameters to be passed between address spaces.

3.4 Software Defect Detection Schemes

There are several software based schemes to implement capability-based addressing and object-based systems on conventional hardware. These systems interpret capabilities as addresses in software. Although very interesting systems result, they are less relevant to this research. However, this research is directed toward efficiently detecting software defects within a program and less concerned with providing a general protection and sharing mechanism for building computer systems. To that end it is interesting to consider systems which attempt to detect software defects within a program. This section presents two such systems which use software techniques applied to the program to detect or prevent memory access errors. Section 3.4.1 briefly presents Purify, the defect detection tool used in Chapter 2 to survey software systems. Section 3.4.2 presents Safe-C, a source-to-source translator aimed at eliminating the possibility of stray memory accesses. Neither of these systems

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

attempt to be a general solution to protection to base a computer system architecture on. Instead they focus on the problem of building reliable software. Both are excellent tools for debugging software, but rely on added instructions to check memory references. Such checking code makes these systems too inefficient to be practical. Purify and Safe-C are examples of object-code insertion and source-code insertion, respectively.

3.4.1 Purify

The basic operation of Purify was described in Section 2.2.3 on page 10. As noted, Purify instruments each byte of program data with two tag bits. The instrumentation process is applied to the object code of a program. For this reason Purify can instrument software without having access to the source code. This implies all code is checked, including libraries. As a result of using object code insertion technology, Purify is relatively independent of the programming language but at the same time very processor dependent and extremely difficult to port. Purify maintains the state in tag bits by intercepting (installing a jump to the checker routine) all memory access instructions in the object files, including the memory allocation routines.

Space for the tag bits must be allocated in the instrumented program and represents a 25 percent overhead when considering just the program data. If data is about the same size as the code for a program this would represent a 12.5 percent overall expansion of memory requirements for the program. But the checking code would need to be added to this overhead calculation. In performance it has been published that instrumented code averages 2.3 times longer in execution time and has a bounded upper limit of 5.5 times normal. However, experience during the research for Chapter 2 indicated an average execution time of about 5 times normal.

Purify has its limitations. It does well on memory allocated from the normal heap space but not as well on other types of data. For example, array bound errors are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

not detected in static or stack memory. Purify can only determine if a memory location is currently active in the context of program execution. It cannot detect a dangling pointer reference to memory which has been re-allocated for another purpose. Purify also cannot detect an array bounds error which is far enough out of bounds. Finally, accesses using corrupted pointer values which happen to point to active memory for the program cannot be detected.

3.4.2 Safe-C

Safe-C is a software preprocessor which transforms C software into an instrumented C program capable of detecting pointer and array access errors [1]. Safe-C transforms programs at compile-time to use an extended pointer representation called a *safe pointer.* A safe pointer contains object attributes as well as the address of the object. These attributes include the current value of a pointer, the base address of the object, its size, its storage class, and a capability. The current value can contain any bit pattern without restriction until the pointer is used (de-referenced) to access memory. Safe-C inserts de-reference checking code to make sure only valid accesses are allowed. Using this scheme Safe-C can check most accesses to memory through pointers or arrays (implemented in C with pointers).

A de-reference check first checks the storage class and if it is a global object the access is valid. Otherwise the capability associated with the pointer is checked. The capability will determine if the associated address has been released to the free space or re-allocated for another purpose. If the memory holding the object is still valid then the access is valid. Once an access is determined to be valid, a check for a valid address within range of the object will be made using the object base and size values. Only when an access is determined to be to a valid object and within range of the object will the access be performed.

Safe-C differs from Purify in that it operates on source code which makes the translator language dependent but not processor dependent. Safe-C improves coverage

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

in several areas as compared to Purify. Safe-C can determine any array bounds error while Purify can only detect nearby bounds violations. Safe-C can also determine all dangling pointer de-references while Purify can only make such determination while the memory remains in the free memory pool and is not re-allocated. Purify cannot make these checks because it does not have information about the context in which these accesses are made, a limitation inherent in working at the object code level.

Safe-C has its problems as well. Some pointer manipulations cannot be detected by the translator. An example of such is when a union is composed of an integer and a pointer. Safe-C cannot maintain a valid safe pointer representation while the value is operated on as an integer. Safe-C cannot be applied to code which is not available in source code form, such as the libraries the application is linked to. Purify does not have this limitation.

Software which has been transformed by Safe-C can expect 100 percent expansion of its data space requirements on average. Execution times for programs range from 2.3 to 6.4 times normal. These overhead penalties are caused by a 450 percent overhead in the representation of pointers along with the added code required to manipulate, maintain, and check these pointers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 4

Logical Object Tagging Architecture

This chapter will present the Logical Object Tagging Architecture (LOTA). It is presented as a stand-alone architecture (architecture reference) but in fact is intended to be a portion of a general purpose processor architecture. Chapter 5 will analyze an implementation of LOTA as part of such a general purpose processor. Chapter 6 will revisit the architecture presented here to explore alternatives to design decisions, provide some insight into tightening the performance bounds, and present interesting future directions for the architecture.

4.1 LOTA Reference

4.1.1 Overview

The design of the Logical Object Tagging Architecture (LOTA) has only two goals:

1. Assure objects have exclusive access to their data.

2. Achieve the first goal through mechanisms efficient enough to be used during the entire object-oriented software life cycle.

Chapter 3 presented several architectures which are capable of satisfying the first goal. None of these architectures succeeded with the second goal. Part of the problem with these earlier systems was that they tried to make the hardware too knowledgeable about objects: where they were, how big they were, what permissions they had to call other objects, etc. This led to a large amount of information for the processor to maintain and large resource requirements inside the processor in an attempt to apply this information efficiently. LOTA attempts to avoid the same pitfalls by enabling the processor with one simple piece of knowledge about objects: data belongs to a spe-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

cific object which has exclusive access rights to it. LOTA applies two novel ideas to reach its goals. The first is that data, in memory, is tagged with the identity of its owner (which is an object). No other system has tagged data for ownership. The second is that these tags are checked in the cache. No other system has used the cache to perform access rights checking.

The block diagram for the LOTA is presented in Figure 21. LOTA is actually an

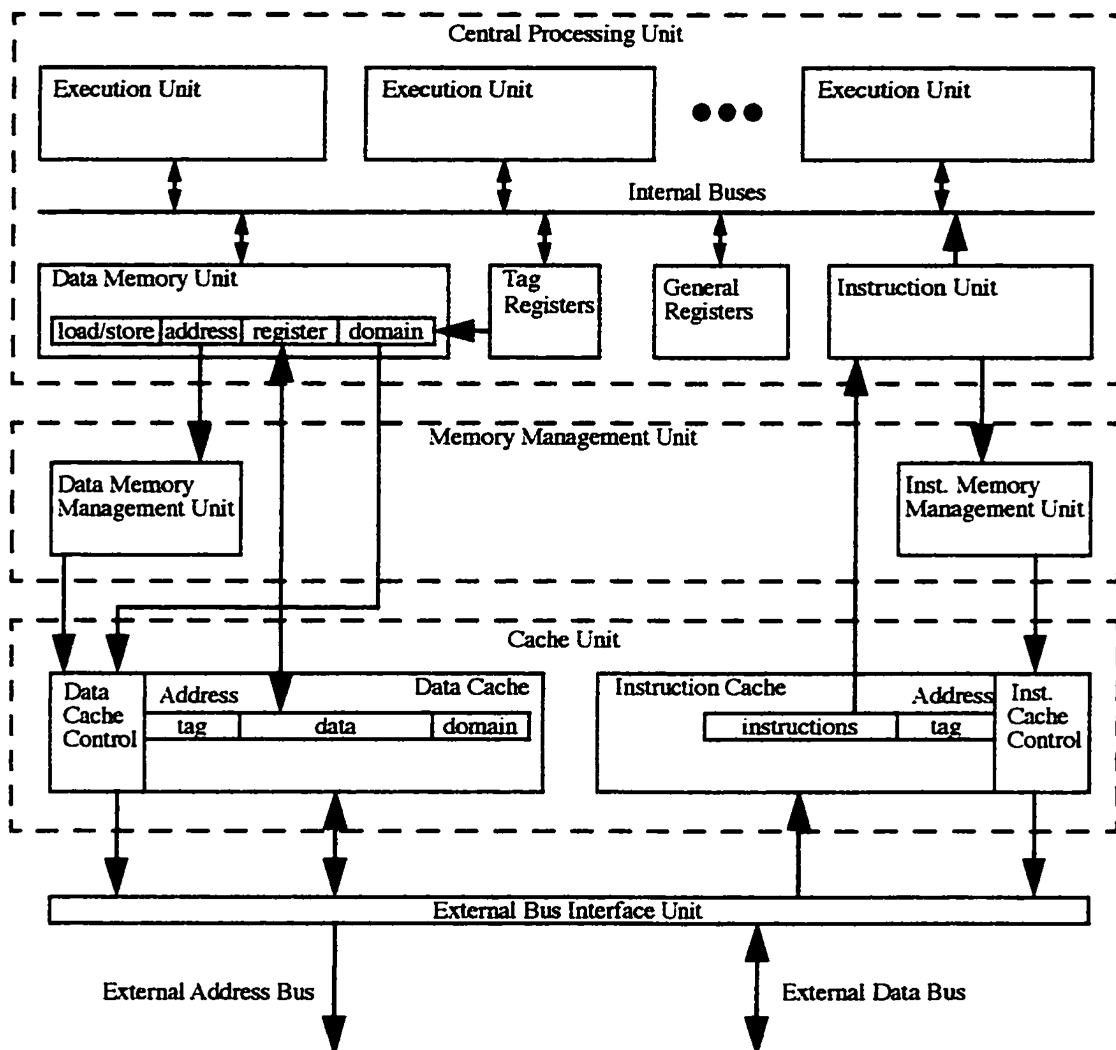


Figure 21 Logical Object Tagging Architecture

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

architecture *extension* which can be grafted onto many general purpose processor architectures. For simplicity, LOTA is presented as part of a Reduced Instruction Set Computer (RISC) architecture. The block diagram shows instructions and data passing through separate memory management units and cache units. This emphasizes that LOTA derives its benefits from enhancements to the data path and leaves the instruction path unchanged.

The central processing unit, as shown in Figure 21, contains an instruction unit, several execution units, a data memory unit, a tag register file, and a general register file. The instruction unit is responsible for fetching instructions and scheduling their execution by one of the execution units. The only instructions in LOTA which access data memory are the *load register from address* and *store register to address* instructions. These instruction types are executed by the data memory unit. In a processor without the LOTA enhancements, load and store instructions can be specified by the triplet $(load/store \cdot register \cdot address)$. The first element specifies the direction, the second specifies which register, and the third specifies what address. The address itself may be calculated (i.e. the sum of two general registers), but if so it must be done at the time the instruction is issued to the data memory unit. As shown in the data memory unit of Figure 21, LOTA uses a 4-tuple $(load/store \cdot register \cdot address \cdot domain)$ to specify a load or store operation. The domain element specifies which object was executing when the load or store instruction was issued. The domain value is supplied by the tag registers which are maintained by the compiler.

The data cache maintains a copy of the most recently used data from main memory. The cache copies of data are tagged with the main memory address used to fetch them. All load and store instructions operate on this copy of memory and leave the coordination with main memory up to the data cache controller. The data memory unit passes the address to the data memory management unit for translation before it is passed on to the data cache. The data cache must first make sure a copy of the data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

from main memory is in the cache. The load or store instruction can then complete by transferring the data between register and (cached) memory. In LOTA, the domain field in the cache is checked against the domain element in the data memory unit 4-tuple $(load/store \cdot register \cdot address \cdot domain)$ to make sure the access is allowed in the context of the executing object.

Figure 22 can help clarify how object-based protection is accomplished. The compiler stores the virtual address of an object in a tag register. Load and store instructions use this value as the domain element of the 4-tuple $(load/store \cdot register \cdot address \cdot domain)$. Data in the cache contains two additional fields. The address tag field contains the physical address used to copy the data from

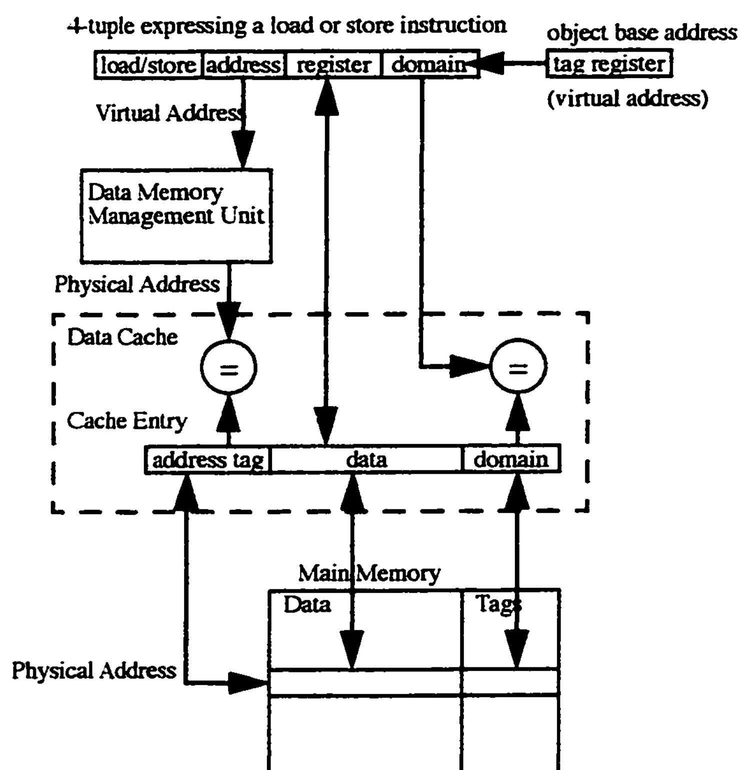


Figure 22 LOTA Object-based Protection

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

main memory. This field is used to match the translated address of the load/store instruction. As part of the cache copy operation the main memory tag is copied, without interpretation, into the domain field. The cache uses the domain field to check the access rights of the load/store operation. As can be seen by this description, logical (virtual) addresses of objects are being used as tags in this architecture, hence the name Logical Object Tagging Architecture.

4.1.2 Programming Model

LOTA presents two programming models. The user programming model, as shown in Figure 23, is a subset of the supervisor programming model. The user programming model consists of three tagging registers each of which have special functionality associated with its use. These registers are large enough to contain a logical address as may be generated by a program. Register t1 represents the currently executing object (alias for this register will be $t_{current}$) and all access to data memory is performed in the context of the object indicated by this register. Register t0 represents the *next* object (alias t_{next}) to be executed and is used to update the tag in memory on certain data memory operations. Register t2 is the *previous* object (alias $t_{previous}$) to have been executing.

When one object calls another in a different protection domain these registers are rotated as $t0 \rightarrow t1 \rightarrow t2$. This implies the current object becomes the previous while the next object becomes the current one. Register t0 is not changed during this

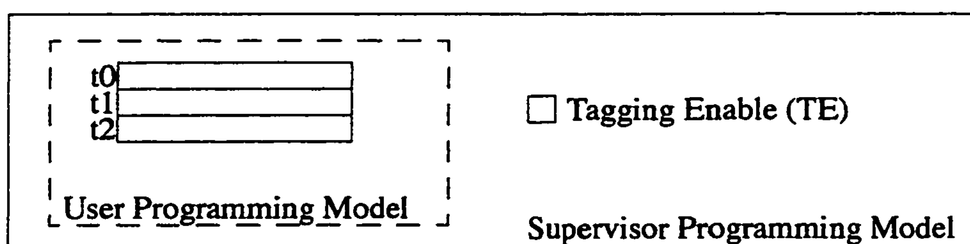


Figure 23 LOTA Programming Model

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

operation.

When an object returns to the one which called it the registers are rotated in the reverse direction as $t2 \rightarrow t1 \rightarrow t0$. This implies the current object again occupies the next object position while the previous object again becomes the current executing object. Register t2 is not changed during this operation.

Use of the tag registers is restricted to protection domain definition and enforcement. They cannot be used as general purpose registers to hold computations or addressing information. Transfers into and out of these registers can only be performed as register-to-register operations and cannot be the target or source of memory load or store operations. These registers are initialized during normal program initialization and are maintained by the compiler using transfers between general registers.

The supervisor programming model includes the user programming model as well as a single tagging enable (TE) bit. This bit is used to indicate if the user process will utilize the tagging protection mechanisms. This bit has its state distributed to all other parts of the architecture which require such knowledge for proper support of both tagging and non-tagging processes.

4.1.3 Addressing Modes and Instruction Set Summary

There is only one addressing mode for transferring information into or out of the tag registers. This addressing mode is *control register addressing* and specifies transfer between a general register and a tag register. For instructions which do not have a tag register as its target or source operand, LOTA places no restriction on addressing modes. In this discussion *S* will be used to denote a source register and *D* will be used to denote a destination register. For example, *tD* denotes the destination tag register and *rS* represents the source general register. *Domain* will be used to denote which object owns a memory location and will be described in more detail in Section 4.1.4.

Load Tag Register. The *load tag register* instruction transfers a designated source

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

general register in the processor to the designated destination tag register.

$$rS \rightarrow tD$$

Store Tag Register. The *store tag register* instruction transfers a designated source tag register in the processor to the designated destination general register.

$$tS \rightarrow rD$$

Enter Protection Domain. The *enter protection domain* instruction jumps to a subroutine, which causes the flow of program control to be changed, while at the same time causes the tag registers to be rotated. The effect is to make a call in a new protection domain.

$$t0 \rightarrow t1 \rightarrow t2$$
$$target\ address \rightarrow program\ counter$$

Return from Protection Domain. The *return from protection* domain instruction returns from a subroutine, which causes the flow of program control to be changed, while at the same time causes the tag register to be rotated. The effect is to return from a call to a previous protection domain.

$$t2 \rightarrow t1 \rightarrow t0$$
$$return\ address \rightarrow program\ counter$$

Load Register. The *load register* instruction loads a general purpose processor register with data from the specified address. The operation is only allowed if t1 matches the domain associated with the data.

$$if(t1 = domain) \Rightarrow rD \leftarrow [address]$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Store Register. The *store register* instruction stores a general processor register into the specified address. The operation is only allowed if t1 matches the domain associated with the data.

$$if(t1 = domain) \Rightarrow [address] \leftarrow rS$$

Store Register with Tag Update. The *store register with tag update* instruction stores a general register into the specified address and at the same time updates the domain field of the corresponding data with the value in t0. The operation is only allowed if t1 matches the domain of the associated data or if the operation is to a free stack location.

$$if(t1 = domain) \Rightarrow [address] \leftarrow rS, domain \leftarrow t0$$
$$if(free\ stack\ address) \Rightarrow [address] \leftarrow rS, domain \leftarrow t0$$

Update Tag. The update tag instruction updates the domain field of the corresponding data with the value in t0. The operation is only allowed if t1 matches the domain of the associated data or if the operation is to a free stack location.

$$if(t1 = domain) \Rightarrow domain \leftarrow t0$$
$$if(free\ stack\ address) \Rightarrow domain \leftarrow t0$$

4.1.4 Instruction and Data Caches

LOTA implementations contain a cache memory in the processor to speed access to data and instructions which exhibit locality. This cache is physically tagged so that it does not need to be flushed on a process switch. How the cache is organized for a specific implementation is not specified in the architecture. The cache may be set associative or direct mapped. The cache may be split between data and instructions or

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

it may be unified. These details can be chosen in each implementation.

As Figure 24 shows the cache organization is based on a set of N adjacent data

Figure 24 Cache Organization

words, a state field, an address tag, and a domain field. Taken together the fields make up a cache line. These cache lines are duplicated as necessary to implement the cache organization desired for an implementation of LOTA.

The state field consists of bits necessary to maintain cache coherency in a multi-processor environment. They may also be used for cache coherency in a system design where the processor is not the only device which moves data around in the system. For cache coherency schemes which use the MESI (modified-exclusive-shared-invalid) model, it would require two state bits to represent the four states in the model.

The address tag is the physical address of the memory location which the line contains. Using the physical address to tag the cache line indicates that address translation, from virtual (program) addresses to physical memory addresses, must take place before a cache operation may be performed. This physical address tag may be reduced in size by the number of bits needed to address each individual data byte in the cache line. These bits identify a particular byte within the line while the address tag indicates the invariant part of the address.

The domain field indicates which object owns this physical piece of memory. Program (virtual) addresses are used as values for the domain field. Since addresses in LOTA specify a byte location, but memory is organized as words, this makes one or more least significant address bits unnecessary. The first of these unnecessary bits (the least significant bit) is use to indicate if the data in the line is global data as shown in Figure 25. Global data is accessible to all objects and therefore may bypass (match on

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

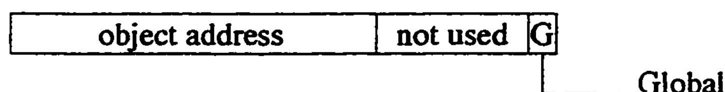


Figure 25 Domain Format

all) domain comparisons. The number of bits not used may vary depending on the implementation. Bits that are left unused may be utilized for other functions.

4.1.4.1 Management of Domains in Cache. The *load register* and *store register* instructions are issued in the context of the currently executing object. As such they are assigned the tag contained in t1 (i.e. capability to access the address) as a ticket to present to the cache controller for accessing a particular cache line. This operation is illustrated in Figure 26.

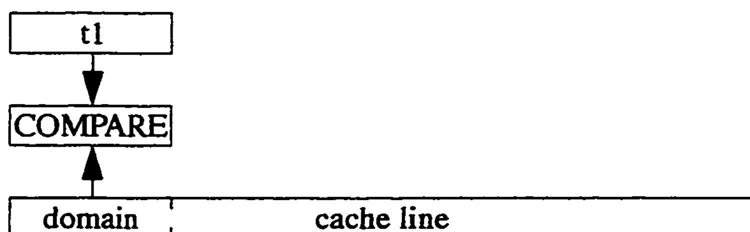


Figure 26 Domain Management for Load and Store Instructions

Store register with tag update and *update tag* instructions are issued in the context of the currently executing object. As such they are assigned the tag contained in t1 (capability to access the address) as a ticket to present to the cache controller for accessing a particular cache line. They are also assigned the tag contained in t0 for the purpose of updating the domain. The domains are managed as illustrated in Figure 27. Recall that the value in tag register t1 represents the currently executing object. The domain field in the cache line indicates which object owns this cache line. These two values must match for the load or store operation to be allowed. Otherwise, if the data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

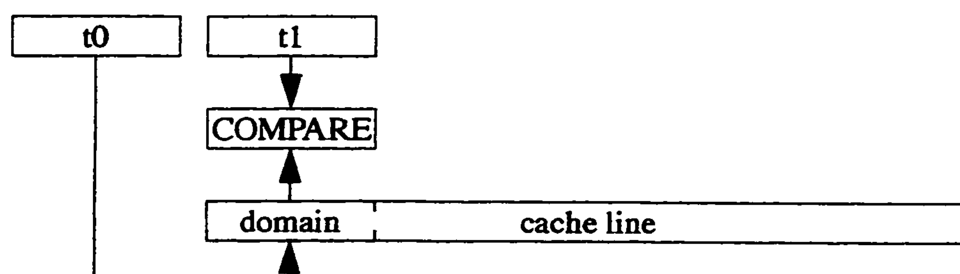


Figure 27 Domain Management for Update Instructions

is not flagged global, a processor exception is initiated. If the operation is allowed, the value in tag register t0 is used to update the domain field in the cache line. Recall that the value in t0 represents the next object to be executed. In this context, *next* refers to the next object to access the associated data. In the normal case t0 and t1 hold the same value. However, changing t0 to another value allows an object to pass data it already owns to another domain, effectively giving the associated memory away. This mechanism will be utilized for passing arguments, sharing memory, and managing tags in the heap space.

4.1.4.2 Load/Store Instruction Ordering. As can be seen in the discussion in Section 4.1.4.1 data memory access must be performed with respect to t0 and t1. In advanced computer architectures instructions have the potential to be issued or allowed to complete out-of-order with respect to the program. LOTA places the restriction that instructions which update the domain field place a barrier which inhibits such reordering of data memory access instructions when the target address is to the same cache line. In other words, the *store register with tag update* and the *update tag* instructions create a barrier in the processor which the *load register* and *store register* instructions cannot cross in the instruction scheduling process. This barrier applies only when the target cache line for a *load register* or *store register* instruction target the same cache line as the instruction creating the barrier.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

4.1.5 Memory Organization

LOTA puts an unusual twist to memory organization. As Figure 28 shows it is a very familiar organization in most respects. Data in memory is organized as cache lines. This is a logical organization emphasizing that the unit of transfer between the processor and memory is the cache line. Programs operate on data in cache while the processor manages the working set of cache lines with transfers between memory and cache. The actual data transfer operation may be a single move of all data in parallel, or may be multiple moves of individual words which make up the cache line. If multiple moves are required, then the tag field will be transferred first followed by the normal algorithm of transferring words in lines when tagging is not enabled.

The twist in LOTA is the additional tag bits. These bits are an *optional* part of the memory system. The tag bits must be present to benefit from LOTA, otherwise the system operates as a normal non-tagged machine. Data transfer is requested by the processor but actual control of the memory cycle is generated as part of the memory system. The processor passes control information to the memory controller. Part of the control information is the flag indicating that object tagging is enabled for this transfer. This flag is the state of the tagging enable (TE) control bit described in Section 4.1.2.

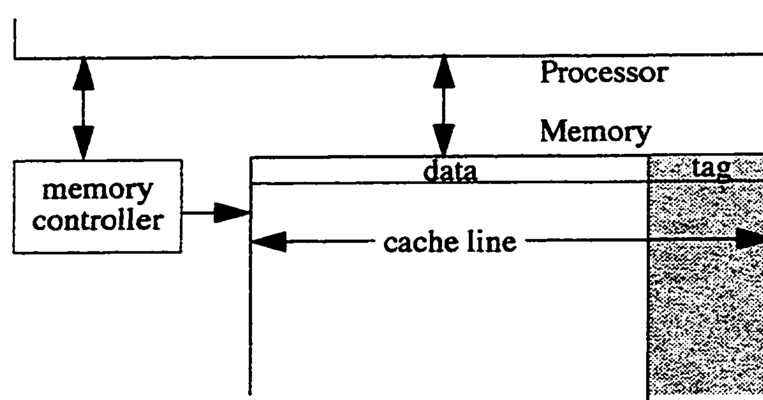


Figure 28 Memory Organization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

When the TE bit is set data transfers include the data for the cache line as well as the tag. The tag bits are transferred between the domain field in the cache line.

As presented LOTA wastes the tag bits when they are not used. This is true for instruction memory as well as all memory in programs which do not utilize the tags. There are methods to reduce this waste but such techniques would complicate the discussion of the architecture. A method to use out-of-line memory (associating tag memory to data memory at run time) will be discussed in Chapter 6. LOTA also appears to complicate the implementation of a compatible family of processors by limiting the selection of cache line size between family members. One possible solution to this problem is to specify in the architecture a method for cache line size to be discovered at run time. The run time system would then dynamically configure a program to the cache line size. An alternative approach would require each implementation to subdivide cache lines at specific levels each containing their own tag.

4.2 Programming Considerations

In LOTA, the register sets are considered to be the exclusive resource of the compiler and data in registers are not protected.

4.2.1 Domain Crossing

The most critical part of LOTA is the efficiency in which protection domains may be changed. LOTA has a very simple instruction set for dealing with tags and crossing protection domains. The process to cross a domain consists of loading register t0 with the object to be called and executing the *enter* instruction to call a routine in the target object. The process is complicated by the fact any previous tag value in t2 must first be saved. Figure 29 shows a pseudo-code fragment which would be suitable for an object call. This code fragment does not deal with arguments to be passed. Lines 1 and 2 move the previous object's tag in t2 to the stack since t2 will be destroyed in the procedure call. Lines 3 and 4 initialize t0 with the object identifier for the procedure

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

```
Domain Switch:
1   move t2 to rD
2   move rD to stack
3   load target object address into rD
4   move rD to t0
5   move procedure address to rS
6   enter rS
```

Figure 29 Domain Switch Pseudo-code

to be called. Line 5 moves the address of the procedure to be called into a register and line 6 enters the new procedure causing a domain switch. This pseudo-code was kept simple and each instruction should have a one-to-one translation into the assembly language of most modern processors.

A closer look at the code in Figure 29 will reveal which instructions are required in any procedure call and which ones are a result of overhead for support of the tagging mechanism. Lines 1 and 2 save the previous value. These are surely overhead operations. However, each procedure will only have to store the calling procedures return tag once. Once saved any number of procedure calls can be made without restoring this value. Procedures which do not call other procedures (leaf routines) do not need to save the previous tag at all. Line 3 loads a register with the base address of the object to be called. In C++ the pointer to an object's data structure is an implicit argument and hence this instruction would need to be performed in any event. Line 4 moves the object base address to the tag register t0, an overhead operation. Lines 5 and 6 implement the object procedure call in exactly the same way a normal procedure call would be made. In this minimal example, three of six instructions are overhead and two of these could be amortized across all procedure calls made while leaf routines do not need them at all.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Consider the C++ code fragment shown in Example 14 where *A* is an object. In

Example 14. C++ Method Call

```
A.fun();
```

this code fragment an object method call is demonstrated. The object method *fun* is called with no arguments. Consider the translation of the C++ statement into a likely C statement as the CFront C++ to C translator would produce. Example 15 shows the

Example 15. C Implementation of a Method Call

```
_A_fun(&A);
```

likely statement to be generated. Notice an argument has been added in the C implementation and it is *&A*, the address of the object. This is the same value LOTA uses to tag the data which the object owns. Passing the base address of the object as the first argument makes sense as there may be many objects of the same type in the system all sharing the same routines to manipulate their data. Each method (object procedure) invocation must therefore identify which object implementation is to be operated on. LOTA simply needs this argument to be copied to tag register t0 before the method call (subroutine call) is made. Since the first argument to a subroutine is normally passed in a register the only additional work in LOTA is to copy that register to t0, a single register-to-register instruction.

4.2.2 Stack Management

Temporary data in LOTA is maintained on a stack. The stack is a special area which needs to be shared between procedures, but yet stack space is controlled by the compiler. Languages like C and C++ have no notion of a stack. Stack space gives the compiler an efficient temporary storage location to enable it to pass arguments, save register values that need to be re-used, and for allocating temporary procedure variables.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

For an example of the problems associated with the stack consider Figure 30. In

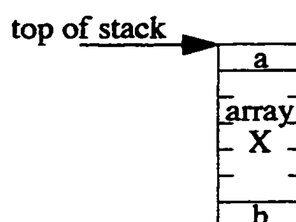


Figure 30 Stack with array

this figure an array, X, of 5 elements is stored in a stack and on either side of the array is another program variable, *a* or *b*. Now consider what the programming error in a C source code statement such as $X[5] = 0$ would cause. An adjacent program variable is overwritten. Although the stack is maintained by the compiler, simple programming errors could undermine the integrity of such values. A major goal of LOTA is to limit these types of errors from propagating to unrelated objects. Since the adjacent value could possibly belong to another object, LOTA must protect temporary data on the stack.

Another difficulty with stack management is the required sharing of data when passing large amounts of temporary data. Consider the code fragment in Figure 31. In

```
#typedef struct Btype {
   int x[100];
};

main()
{
   Atype a;
   struct Btype b;
   a.fun(b);
}

a::fun(struct Btype b) {
   for(int i=0; i<100; i++) b.x[i] = 0;
}
```

Figure 31 Structure passing code fragment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

this code fragment it can be seen that one routine is calling method *fun* in object *a* and passing the structure *b*. If the structure b is large, then it must be passed on the stack. This implies the calling routine must copy the structure to the stack and then pass it to the method *fun* in object *a*. LOTA must therefore provide for the ability to create temporary space which can be passed to another protection domain. This operation is expected to be frequent and therefore must be very efficient.

LOTA provides two simple capabilities to make this sharing of stack space in a protected method very efficient:

- Stack space can be claimed as needed.
- One object may claim stack space for another object.

In order to claim stack space as needed a procedure only needs to use the *store with tag update instruction* to store its temporary values on the stack. If the locations where these temporary variables are stored are marked global (global in the stack indicates the space is free) then the operation will be allowed. Figure 32 demonstrates

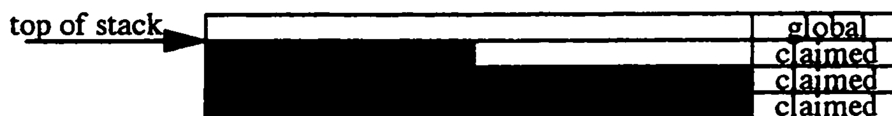


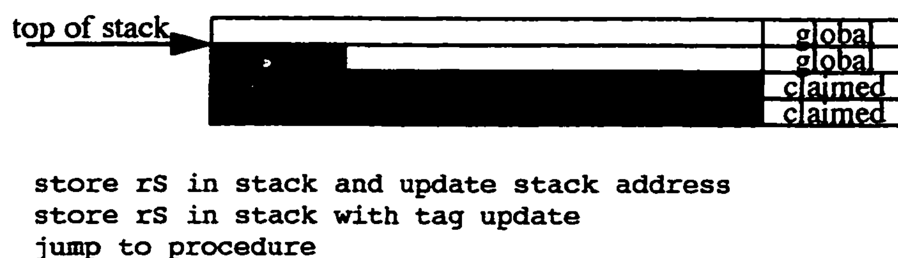
Figure 32 Stack claim operation

how the stack is claimed. In the figure stack is illustrated as cache lines and the shaded part indicates the example area claimed by a procedure. Each time a *store with tag update* instruction is applied to a cache line marked global it claims the space according the to value stored in t0. There is no overhead involved in claiming stack space.

When a procedure is finished using its stack space it must return that space. This requires one *update tag* instruction for each line of space it claimed. The instructions executed to return stack space are overhead operations inherent in LOTA.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

There are several ways to share data on the stack between procedures. Space could be claimed by the calling procedure and then given to the called procedure prior to making the call. Another possibility would be to store the values to be shared on the stack without claiming them. The claim operation could then be performed by the called procedure as it sees fit. The method described here will be a compromise between these two possibilities. The method is demonstrated in Figure 33. This figure




```
store rS in stack and update stack address
store rS in stack with tag update
jump to procedure
```

Figure 33 Stack claim for argument passing

demonstrates a partially completed stack frame to be passed to another procedure. The last line of this stack frame has yet to be completed. The pseudo instruction sequence shows how the last two values might be saved on the stack before a procedure call is performed. The second to last value is stored with a normal store instruction leaving the space it occupies as unclaimed stack space. The final value saved on the stack uses the *store with tag update* instruction to claim the space. Assuming t0 has been initialized with the domain value of the object which the procedure to be called is part of, then this stack frame will be owned by the called object.

Just as before, this newly claimed stack space must be returned. But in this case it is up to the called procedure to free the space claimed on its behalf. This demonstrates that claiming stack space in any case is essentially free of performance penalties. However, to return the space requires one memory store operation per line allocated. This overhead will be evaluated in the analysis presented in Chapter 5.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

4.2.3 Heap Management

LOTA can handle the management of heap space in several ways. The method presented here will be a straight forward and simple method. All heap space begins existence with tags initialized as owned by the heap space manager. When a call is placed to the heap space manager it allocates the space required, in terms of cache lines, and then updates the tags with the corresponding object identifier. This represents an overhead of one write per line allocated. This overhead must be repeated to return the space back to the heap manager. This overhead will be evaluated in the analysis presented in Chapter 5.

4.3 Qualitative Analysis

4.3.1 Defect Detection

LOTA primarily uses the encapsulation of object data as the means to detect software defects. This is the same as all the capability-based and object-based systems presented in Chapter 3. Several important defect types can be trapped by object-encapsulation.

Array Bound Access Errors. Array bound read and array bound write defects are not allowed to violate object boundaries. All such errors which attempt to violate object boundaries are trapped immediately. Array bound write errors have the potential to corrupt data which belongs to a different object. When defects are propagated in such a manner they are particularly hard to trace to their root cause. LOTA traps all such potential data corruption defects and will identify exactly which object needs debugging.

If objects are chosen carefully to place arrays inside their own object, or if compilers can lay out the data to reserve adjacent memory as non-allocated space, LOTA can be used to detect a finer granularity of software defects. Unlike Purify, LOTA has no restrictions on where arrays are located for its protection mechanism to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

apply. Purify can only detect array bound access errors when the array is allocated from heap. In fact, Purify can only detect if the memory being accessed is currently active but has no idea what object it belongs to. This in inherent in its technology application of object code insertion which has no information about the actual structure of the program. If an array bound read or write error is far enough off course to be a valid address of an object adjacent in memory the access would be allowed.

Free Memory Access Errors. Free memory read and free memory write errors are completely eliminated with LOTA. Since all memory returned to the heap is immediately marked as belonging to the heap space manager, all accesses to this memory will be trapped. Purify can detect accesses to the free space as well. Safe-C has some restrictions on use of pointers. Converting pointers to integers and integers to pointers, as is often done in C programs, can cause problems with Safe-C's ability to maintain state of a pointer. When these restrictions are not violated then access to memory returned to the heap is detected. Even still, Safe-C adds a capability field to the pointer context to indicate the object pointed to has been freed. A significant amount of software overhead is incurred to maintain this structure. Systems based on capability-based addressing can easily detect access to free memory.

Pointer Defects. Two closely related defect types are the dangling pointer defect and the pointer corruption defect. In the dangling pointer defect a pointer remains active in a program after the memory it points to has been returned to the free space. The corrupted pointer defect as its name implies is the problem of a pointer to memory being mistakenly altered, but yet remains within the address space of the program. Both of these errors have an enormous potential to cause memory corruption in unrelated areas of the software and are extremely difficult to debug. Capability-based systems can ordinarily catch such problems but must use careful design. When capabilities and the address they point to are kept together and allowed to be copied then some care must be taken in software to avoid valid capabilities to point to objects

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

which have been destroyed. When capabilities and the memory they point to are kept in separate data structures, such as in the iAPX432 or System/38, duplicate capabilities for the same segment all point to the same segment descriptor. Such systems only need to mark in one place the object is no longer available. Even so, such systems must take care when the names of capabilities are reused. System/38 avoids this problem by having such a large address space and managing it with the policy that addresses for segments will never be reused.

In LOTA any access to memory must be to memory which is currently owned by some object. Any access to free space for any reason will be denied. There is no opportunity to have a valid capability to an address in free memory. In addition, if memory is assigned to another object it will still be outside the scope of access for any previous object that owned it. There is only one case where LOTA does not detect dangling pointer or corrupt pointer access errors, the case where the memory it points to is assigned again to the same object it was before. It is impossible for LOTA to tell if such a memory access is intended. However, the defect is still limited to within a single object.

Safe-C uses the same mechanism described for detecting free space access errors to detect corrupt or dangling pointer accesses. Safe-C has the advantage that it can determine this fact even when the memory is used again by the same object. For Purify it is only possible to know if the reference is to a valid memory location. If the defect causes an access error to free space it can be detected, but if the memory location is active by any object in the program then the access will be allowed. To help detect dangling pointer defects Purify adopts the memory allocation policy to delay for as long as possible the allocation of a memory block set free. Purify has no mechanism to detect corrupt pointer defects unless they accidentally point to a memory location not active at the time the pointer was used.

Uninitialized Memory Read Errors. Uninitialized memory read errors were the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

most frequent reported in Chapter 2 for the survey of software defects. There errors were the least likely to cause a program failure and are more closely related to bad programming practices contributed to by the poor character handling facilities supplied by UNIX and libraries. Nevertheless, use of uninitialized memory can cause variation to specification for a program and are an important type of error. LOTA only provides limited resources to be applied to this problem as presented so far. Any read of variables outside the domain of the object is flagged. If the uninitialized memory is used as a pointer, or in the calculation of a pointer value, then it is likely to point to memory outside the domain of the object and be trapped by LOTA. This is the same limited capability as all the other systems presented in Chapter 3 with the exception of Purify. Since Purify maintains the state of a memory location it can detect when that location is read before it is written regardless of which object may be executing. Although LOTA has limited capability to detect uninitialized memory accesses it does have the potential to use tags associated with memory to be used for help in this area. Such possibilities are explored in Chapter 6. As a minimum, LOTA restrains such errors to object boundaries.

4.3.2 Domain Crossing

In LOTA a very simple mechanism is used for the software to convey to the hardware what object is executing and what memory it should have access to. The mechanism is a register update operation. The value which is used to program the register is the base address of the object. To programming languages the base address of an object is referenced by the name of the object. Since the name space in the program is completely understood by the compiler, it can easily manage object-based protection. In other words, a compiler can control protection domains using object names, names from the name space it already controls. There is no indirection and the values used for naming the protection domain is most likely to already reside in a general register due to normal subroutine calling conventions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Purify and Safe-C have no protection mechanism to compare with. But compared to capability-based addressing schemes this eliminates the inherent indirection. System/38 and the iAPX432 required a level of translation to convert capabilities into virtual addresses using a two level lookup table. The inherent indirection required the processor to cache the translations as the lookup was quite expensive. Only when capability translations were cached was domain crossing efficient. In the iAPX432 a new context object must be created for the new domain. Even when pre-allocating these objects it was still relatively expensive to initialize their contents.

The Denelcor Heterogeneous Element Processor (HEP) was truly able to switch domains in the common case very quickly. The common case was designed to be between active threads of a program. But this requires a large number of registers in the processor to hold the state of each domain. There were no mechanisms to load state of a new thread and protection domain in a rapid fashion. HEP would need to be expanded to a large number of small protection domains to be useful in object-based protection schemes.

The most interesting of the capability-based addressing designs is the use of guarded pointers in the M-Machine. Guarded pointers are used to provide a solution to the same problem addressed in HEP, the need to interleave instructions from multiple independent threads of execution potentially from multiple protection domains. The compact encoding of a capability eliminates the indirection of System/38 and the iAPX432 which must first translate a capability into a virtual address of an object. The guarded pointer still does not address the efficiency of one of these threads calling another one in a different protection domain. The penalty of such an operation has the potential to be must less than in the HEP, but there is still considerable overhead in clearing registers of capabilities and setting up shared space to pass arguments.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

4.3.3 Abstract Data Type Implementation

Noted earlier was the fact that capability-based addressing and the programming language concept of abstract data types were solutions to the same problem. The structure of software is modularized to reduce the scope of software defects and reduce the maintenance requirements. Object-based systems propose a complimentary system design implementing both capability-based addressing and abstract data type implementation. LOTA adopts the philosophy stated by Wilkes [45] that capability-based addressing and abstract data types are competitive solutions. LOTA does not attempt to provide a complete understanding in hardware of what an abstract data type is. Instead, it leaves the implementation of abstract data types to the compiler and provides the compiler a simple tool to help assure the information hiding principle is not violated.

4.3.4 Resources

In order to support fast domain changes in the multithreaded architecture of the HEP many registers were required to hold the state of several processes at one time. These resources are very close to the processor and to increase these resources to support many objects active at one time in different protection domains would be prohibitive.

Capability-based addressing systems require large amounts of local storage in the form of a cache to avoid repetitive lookup operations. The cache requirements are in addition to, and in resource competition with, the normal data cache. All of the capability-based addressing systems, with the exception of the M-Machine, presented in Chapter 3 were supported in microcode. The complexity of the operations to generate capabilities and check access rights made this necessary. All of these additional resource requirements occurred very close to the processor, in the form of additional registers, microcode, and cache.

LOTA pushes the resource requirements farther away from the processor. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

processor itself has three additional registers and associated data paths. The cache contains the largest additional resource requirement, there is an additional field for each cache line equivalent in length to a tag register. Actual cache implementations may allow the number of bits in this field to be reduced. In addition to the extra field in a cache line, LOTA requires a comparator to check the cache line domain with that of the currently executing object. If the cache is organized as set associative then this comparator will need to be duplicated by the number of elements in a set (i.e. in a 4-way set associative cache there are four elements to check in parallel). Although this extra field in a cache line may prove expensive, there is no need for other cache structures to support the protection mechanism. The management of tags in the cache causes very little additional logic to be associated with protection. Cache lines are transferred between cache and memory autonomously with respect to protection. If tags are included in the transfer they are moved as if they were data. The greatest requirement for resources in LOTA is the additional tag bits in the memory system. Every block of memory, the size of a cache line, contains an extra set of tag bits of length equal to an address in the processor. But notice the overall resource requirements, the smallest requirement is placed close to the processor where competition is probably greatest. The requirements in the cache are larger. The largest requirement is in the memory system, the farthest point away from the processor. This is where resources are probably the most affordable and easy to make optional.

4.3.5 Stack Management

Fabry suggested the stack be a hardware-managed stack of processor registers [46]. Unfortunately, even though the stack is controlled by the compiler, it is shared with program variables. An array which is accessed with an out-of-bounds index could potentially effect variables on the stack of other objects. LOTA allows normal stack manipulation within a routine. However, stack space used by one object is not accessible by another. As with ordinary programs stack is used as needed, but the act

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

of using this space also claims it. Once claimed it is impossible to access the space without help of the object that claimed it. The stack allows protected sharing for the purpose of passing arguments and return values. The cost of this design is one tag clear operation for each line of stack space set free. For small stack allocations, as they are expect to be, this represents a minimal cost.

4.3.6 Heap

Heap space management will include the added complexity of allocation on cache line boundaries. Each line allocated or deallocated must have its tag updated. For programs which allocate small or infrequent blocks of memory from heap this tag update penalty will be small. Since memory is allocated in terms of cache lines there will be some small amount of space allocated above what is requested. If heap allocations are random in size the expected wasted space will be one half of a cache line per heap allocation. Compare this to the potential wasted space in the M-Machine where heap must be allocated in power of two sizes. In this machine the space wasted, given random allocation sizes, will be 25 percent. Chapter 5 will analyze the tag overhead for heap allocation as well as heap utilization.

4.3.7 Simplicity

Although LOTA adds some complexity in the compiler in order for it to handle tags the rules for tag manipulation are straight forward and simple. The rules are:

- Initialize tags for heap allocation.
- Clear tags for heap deallocation.
- Claim tags for stack allocation.
- Clear tags for stack deallocation.

These rules for managing tags are consistent and are not effected by frequency or size of memory allocated in heap or stack space. Only the overhead of the operations vary. Once the tags are initialized properly the only complications the compiler must deal with are allocation of memory in terms of cache lines and managing the tag registers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

4.4 Segmented Alternative

LOTA uses a limited support in hardware to derive its benefits. So far LOTA has only been compared to other systems which have complete support for system level security or are software only systems to detect software defects. This section provides a comparison of an alternative design which also provides limited hardware support to allow the compiler to control protection domains for objects within a program.

The use of segments is most often mentioned in the scope of providing bounds checks for accessing memory. In this scheme objects or arrays are allocated one to a segment. Figure 34 demonstrates how address generation is performed in a processor using segmentation. A segment is identified by a *base* and a *bound* denoting the start

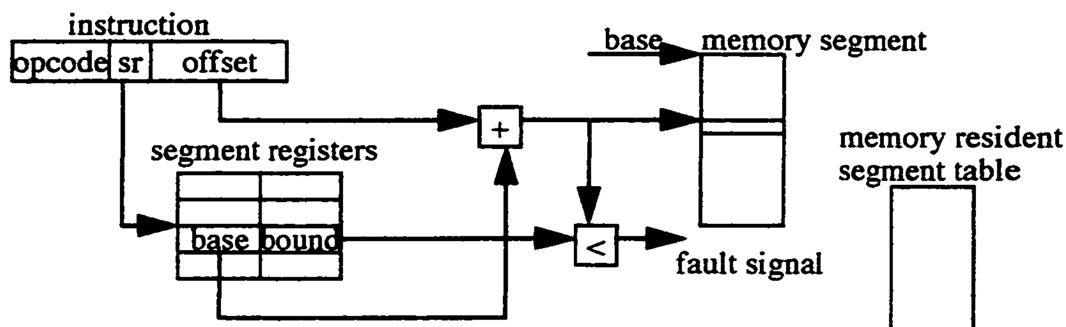


Figure 34 Segment Address Generation

and end address for a segment, respectively. Instructions must specify a segment and an offset to access memory. Segmented architectures customarily contain segment registers which are loaded as needed from a segment table. The segment table contains an entry for all segments which can be reached within a protection domain. Instructions specify the segment register which has already been loaded with the segment information. Each entry in the segment table is a data structure called a segment descriptor and contains all the necessary information about the segment. The processor must make sure the offset specified in the instruction is within range of the segment bound and generate a fault if it is not.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

4.4.1 80386 Processor

The 80386 processor is the first in a family of processors which is capable of 32 bit segment addresses with 32 bit segment sizes for generating 32 bit linear addresses to be used as program addresses for accessing memory [47]. In effect, the processor applies segmented address space management to generate program addresses to be passed on to a traditional memory management unit and cache unit. This segmentation capability has often been discussed in the Internet news groups for its application to array bounds checking and even in the possible application of one object per segment protection domains [A.1]. Before comparing LOTA to a base and bound approach to object-based protection a discussion of the 80386 segmented architecture and its application to object-based protection will be presented.

The 80386 contains six segment registers (sr) which provide an index into one of two segment tables and a flag to choose between the tables. These tables are known as the global descriptor table (GDT) and the local descriptor table (LDT) and are shown in Figure 35. Descriptors in the descriptor tables are all 64 bits long and provide a 32 bit segment base address and a 20 bit segment size limit along with protection information. The processor contains a global descriptor table register (GDTR) to point to the global descriptor table and a local descriptor table register (LDTR) to point to the local descriptor table.

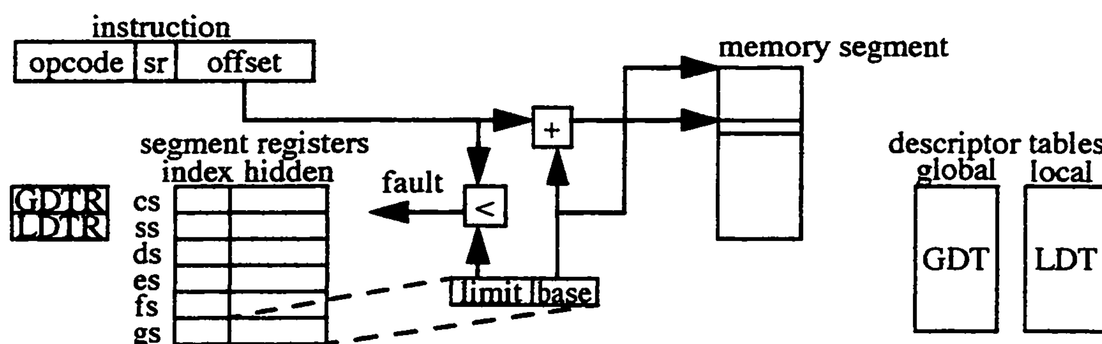


Figure 35 80386 Segment Addressing Model

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Program addresses are generated as shown in Figure 35 and is very similar to that shown in Figure 34. The instruction provides the segment register (sr) to use, or one is implied, and the base is combined with an offset while the offset is compared to the limit. The segment registers may be freely loaded from the descriptor tables by the compiler. However, the segment register is actually maintained by the processor as a visible part and a hidden part with respect to the program. The visible part simply contains the offset into the descriptor table and a flag to indicate which descriptor table, but the invisible part contains the base address of the segment as well as the segment size and some protection information. In other words the segment number used to load the register from the descriptor table is visible to the compiler but the register actually contains all the remaining information about the segment in hidden fields. In this way the processor contains all the protection domain information relevant to access a descriptor as state inside the processor while not divulging the information to the program or compiler.

The 80386 has many limitations in the application of its segmented architecture to object-based protection. First, the descriptor tables can only be modified by system software. They are designed to provide process-based protection by defining the entire address space a process has access to. The table in fact *defines* the protection domain for a process. To add new objects to the system would require operating system intervention, something much too costly. The local descriptor table only has entries for 8,192 entries. If each object required four segments (code, stack, and two data segments) then only 2,048 objects could be active at any time during the life of the program. This is much too limited for the general case.

It is evident that the standard design of the 80386 is meant for process-based protection and would not apply well to object-based protection. Even on a process base using segmentation to support protection has its problems. Compilers must keep loading the segment registers to access different segments. Primarily the segmented capabilities of the 80386 have been used to provide a faster 8086 compatible processor

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

which does not use the protection aspects of segmentation. The segmented model is not well suited to the simple paging protection of UNIX either. In fact the segmented protection mechanism has not been used by any major commercial operating system [48].

4.4.2 Modified 80386 Segmented Architecture

To give a more balanced comparison with LOTA consider the modified 80386 segment addressing scheme in Figure 36. This new model only shows the part of the modified 80386 which is under the control of the compiler. The global segment table still exists but requires operating system help to modify and is therefore not shown. In this modified model the local descriptor table and local descriptor table register have been renamed the object descriptor table (ODT) and object descriptor table register (ODTR) to emphasize they now define the protection domain for one object. The resulting segmented architecture will allow compiler control of segments for object-based protection without operating system intervention and without limitation on the number of objects active in the system.

In the modified 80386 the compiler controls the protection domain by loading the ODTR to point to the table defining the domain for an object and managing the segment register values from this table. On object calls the ODTR must be changed and

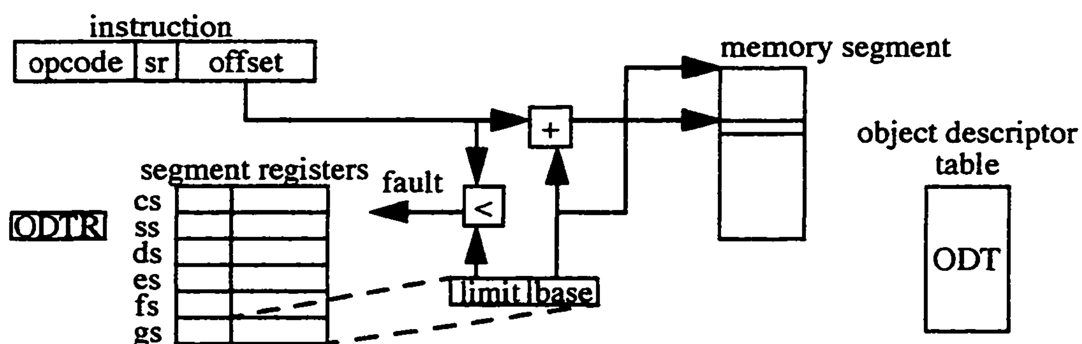


Figure 36 Modified 80386 Segmented Addressing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

any values in the segment registers must be cleared. Notice the similarities to the Plessey 250 where capability registers specified the address space available to instructions while capability segments defined the full range of the available address space. The modified 80386 requires an object descriptor table for each object just as the Plessey 250 required at least one capability segment for each procedure. Just like the capability segments, object descriptor tables will be many and will ordinarily be small. These small tables must be managed by the compiler.

A new *enter* instruction must be defined which changes the program counter and the protection domain at the same time. The operands to the *enter* instruction must therefore include the target address and the target ODTR value. Just as in LOTA, the old value of the protection domain must be saved for reuse on return. If the called routine is not a leaf routine the ODTR must be saved and restored exactly as required of tag registers in LOTA.

When a new protection domain is entered ordinarily the code segment must be changed. In the modified 80386 model it will be assumed the code segment is global and does not require a change. The stack segment must be changed and still allow sharing between procedures for argument passing. This could probably be implemented fairly efficiently by a large global segment with sliding base and bound values.

Up to this point it can be said that handling the LOTA tag registers for protection domain changes is similar in complexity of handling the ODTR in the modified 80386. The stack handling overhead for the modified 80386 probably excels in efficiency as the size of the stack allocation goes to larger values. On the other hand LOTA requires several fewer instructions for changing the protection domain because it has no additional register to clear and load to define addressing capabilities of instructions.

Tags Versus Descriptor Tables. Recall that descriptors are each 64 bit values. Given the average size of an object is 256 bytes [40] the overhead in memory utilization for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

an object with just two segment descriptors would be 6.25 percent. For a LOTA implementation with 32 bit addressing and 256 bit cache lines the overhead would be 12.5 percent. In LOTA this is a fixed cost to implement the memory system while in the modified 80386 design it is a variable cost and reduces the available memory.

Pointer Representation. In the 80386 a pointer is no longer capable of being represented by a single 32 bit quantity. In order to include the segment identifier a pointer must be represented by a 48 bit value. This represents a 50 percent overhead in pointer representation and must be included in the memory utilization calculations. In a RISC design this odd size would itself cause memory alignment problems. The compiler must determine if the segment identifier is already contained in a segment register when the program uses a pointer. The determination itself may take extra instructions and if the segment is not already resident in a segment register then a segment register must be loaded. The instruction will either imply the use of a specific segment register or it will identify a segment register explicitly which means a pointer will have to be loaded in the exact segment register specified in the instruction to use it leaving less flexibility for the compiler to determine register usage at run time.

Descriptor Table Management and Dangling Pointers. The descriptor table must be managed by the compiler. When a new memory segment is to join a protection domain the compiler must allocate new space in the descriptor table. The compiler must therefore maintain the descriptor table in memory space where there is room to grow. The compiler will probably need to maintain the current descriptor table size in a variable or register somewhere to make sure an index specification does not reference an out-of-bounds descriptor location. Once a descriptor is allocated in the table it can never be removed from the table, but must instead be marked invalid to avoid future use when the segment is destroyed. If this rule is not followed then it would be impossible to ever catch a dangling pointer error.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

The Bottom Line. In the final analysis LOTA compares well to a segmented approach to object-based protection within a process. When stack and heap space are allocated in small quantities LOTA compares favorably with segment management. It compares less well for large allocations. Domain crossings, especially when accompanied by small stack spaces, are much more efficient in LOTA. In segmented systems pointer representations are complicated and require segment checks to dereference along with frequent loading of the segment registers. To catch dangling pointer errors a segmented approach must not reuse a descriptor entry in a descriptor table.

The strongest argument in favor of LOTA is its simplicity. The complications LOTA imposes is limited to allocating memory in terms of cache lines, setting the tags, and keeping the tag register up to date. These chores are simple and consistent. The mechanisms are applied equally easy to large or small quantities and there are no additional complications to the number of non-contiguous memory pieces allocated to one object or the program as a whole.

In the case of segmented systems the number of segments allocated to an object requires management in a dynamic table. Each and every object must have such a table. Pointers must be handled as segment identifiers and offsets. Segment registers must be carefully managed. Segment descriptors and pointers do not fit in a normal word for the processor and require multicycle transfers between memory and register.

Defect Detection and Security. Both systems attempt to solve a part of the software defect problem with the same solution, to limit the realm of damage a defect may cause to an object boundary. As such, they both succeed in their goal. In addition, both systems may be capable of sufficiently efficient operation to allow their use over the entire software life cycle. There is one more area that must be considered and that is which one offers a more secure system. Recall in LOTA the only way to effect the protection domain is by use of the tag registers. Once a memory location has been tagged it is impossible to access unless the tag register is set to the correct object value.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

In LOTA setting these tag registers does not require a special privilege. However, in a high level programming language there is no way to accidentally change a tag register. This is because there is no syntax offered in a high level language to allow an instruction to reference the tag registers. The compiler must generate the code to change these registers. LOTA maintains the protection domain in processor registers which cannot be named by the programming language.

Now consider the segmented system. The descriptor table base address is stored in a processor register. It does not actually identify the protection domain directly, but indirectly points to the table which defines the protection domain. The actual protection domain is the values stored in the object descriptor table. The descriptor table is managed by the compiler in memory that is currently accessible by the protection domain of the object. This implies there is a possibility that a software defect at the source code level could lead to memory corruption within the object that compromises the object-based protection mechanism.

The Tag Advantage. There are several classes of software defects which can be caught with object-based protection. But there are other problems which cannot be caught with this solution alone. Since LOTA uses tags there exists additional opportunities to expand their functionality to apply to other types of software defects. Some of these possibilities will be explored in Chapter 6.

4.5 Information Flow

Tags in LOTA convey information about the domain data exists in with respect to objects of a program. The information is used to detect software defects. A standard program must be augmented with instructions to use and maintain tags before this benefit can be derived. This section will analyze the inherent added flow of information between the processor and memory system and any possible consequences.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

4.5.1 Von Neumann Computer

A simple definition of a von Neumann computer is a computer containing a central processing unit (CPU), a memory (store), and a tube connecting the CPU and store. The CPU communicates with store by sending an address down the tube. Also flowing down the tube is data which corresponds to the address. The tube is known as the von Neumann bottleneck [49].

A greater concern than the existence of this tube as an actual bottleneck is the evolution of popular conventional programming languages as high level, complex versions of the von Neumann computer. These so called von Neumann languages require the programmer to think in terms of this tube and operate a word-at-a-time towards the solution to a problem. C and C++ are examples of popular von Neumann languages. Although C++ offers programmers the ability to define complex types with behavior closer to the natural objects in the problem domain, it is still based on the primitive construct of the assignment statement.

LOTA imposes extra complexity in the compiler to maintain the domain information in tags and to manage the processor notion of current domain. However, this complexity does not show through to the programmer. LOTA offers nothing which alters the picture with respect to the von Neumann languages.

4.5.2 Tag Information Flow

Although LOTA does not effect the programmer's view of the computer it does require additional information flow between CPU and store for the program to take advantage of the added protection. This added information flow is a result of the extra instructions the compiler needs to insert into the program to manage tags and domains.

The analysis of added information flow in LOTA can be divided into the inherent added flow and the implementation added flow. The implementation information flow can be represented in clock cycles as the difference between execution times of a program which is compiled with and without protection in LOTA. Although this unit

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

of measure is not in terms of information flow it does represent the penalty associated with the added information flow for its implementation.

In this section only the inherent added information flow for LOTA will be evaluated. For this evaluation the only information flow of interest is the required, unavoidable transfers of instructions and data between CPU and store. In particular the effects of the implementation of the tube between processor and memory are excluded. To exclude this effect can be easily done by declaring the unit of transfer for data to be the cache line which includes the tag. For instructions the unit of transfer is a single instruction.

Using these definitions of transfer units the added information flow in LOTA can be expressed as the sum of tag manipulation instructions and tag only data transfers. There are only two instructions which manipulate tags, the *store with tag update* and the *update tag* instructions. Only the *update tag* instruction results in a tag only data transfer. Therefore the inherent information flow penalty (IFP) in LOTA can be stated as in Equation 1.

$$IFP = 2 \bullet number\ of\ update\ tag\ instructions \qquad \textbf{(EQ 1)}$$

The 2 in Equation 1 represents one instruction transfer and one tag only data transfer between CPU and store.

There are only three places in LOTA where tag update instructions are required. These places are the release of stack space, heap allocation, and heap deallocation. In each of these three places tags must be initialized independent of data transfers and requires one tag update operation per cache line involved. Let stack release in lines be represented by SRL, heap allocate in lines by HAL, and heap deallocate in lines by HDL. Then the information flow penalty (IFP) can be represented by Equation 2.

$$IFP = 2(SDL + HAL + HDL) \qquad \textbf{(EQ 2)}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 5

Analysis

A quantitative analysis is provided for a possible implementation of LOTA. An 88110 processor is used as a base architecture. The 88110 was selected because of the availability of the following items:

- 88110 computer running UNIX System V Release 4
- CFront C++ translator
- Cycle accurate simulator
- Source code for the simulator

In particular the source code for the simulator was required. Gathering the experimental data for this chapter would have been impossible otherwise. Section 5.1 will present the 88110 as a base implementation for LOTA. Section 5.2 will present the experiments including the simulator, software, and measurements. Section 5.3 will present the results and analysis.

5.1 Implementation of LOTA

5.1.1 The 88110

The 88110 is a general purpose microprocessor using the reduced instruction set computer (RISC) design philosophy [50]. A block diagram for the 88110 is shown in Figure 37. The processor has two register files. The general register file supports integer operations while the extended register file supports floating point operations. The processor has several independent execution units which all take operands from one of the two register files and deliver results back to one of these register files. The instruction unit is responsible for fetching instructions, scheduling their execution, tracking resources, as well as processing change of flow instructions. The data unit is responsible for executing all instructions which access memory.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

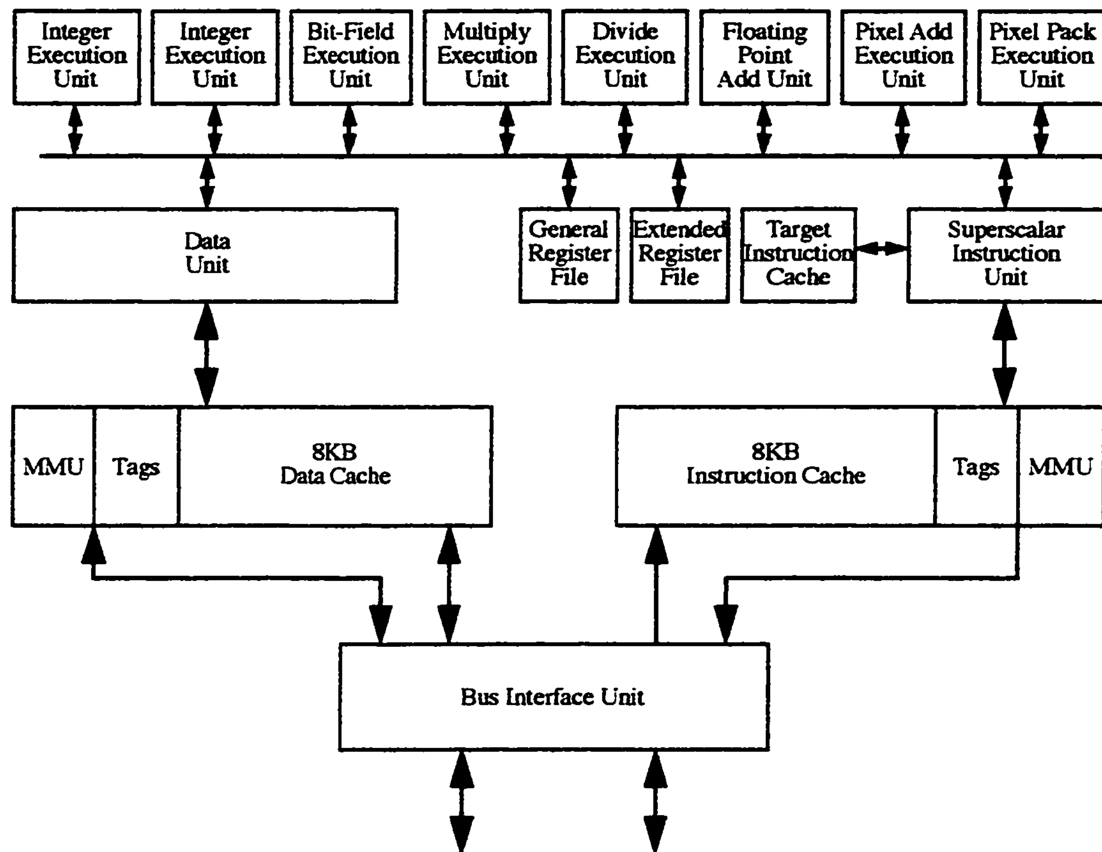


Figure 37 88110 Block Diagram

Internally the 88110 is a harvard architecture with separate paths to memory for instructions and data. The instruction execution unit fetches instructions from the instruction cache and instruction memory management unit (IMMU). The data unit performs all data accesses by use of the data cache and data memory management unit (DMMU). The harvard architecture ends there as both cache units use the same bus interface unit to gain access to the external address and data bus.

The instruction execution unit is in charge of the information flow through the 88110. It fetches instruction pairs from the instruction cache and is capable of issuing two instructions to separate execution units in each clock cycle of the processor. The

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

execution units only operate on data supplied by the register files and only deliver results to the register files, a standard RISC register-to-register design. The only instructions which access data memory are the *ld*, *st*, and *xmem* instructions.

This is a simple description of the 88110 processor and more detail can be obtained from the user's manual [51]. The important traits of the 88110 which make it an excellent target implementation for LOTA is its RISC design, clean architecture, and independent data memory unit for executing all instructions which access memory. These traits lead to an explanation of the design through simple block diagrams that clearly demonstrate the major components and information flow through the processor.

5.1.2 110L

The LOTA implementation of the 88110 processor is called the 110L. The programming model is shown in Figure 38. The model presented is exactly the standard model for the 88110 with the addition of the tag register file. The extended and general register files have already been explained and the fcr63 and fcr62 registers implement the required status and control for floating point operations according to the IEEE 754 floating point standard. The control registers include all the status and control information for the normal operation of the processor. These registers are

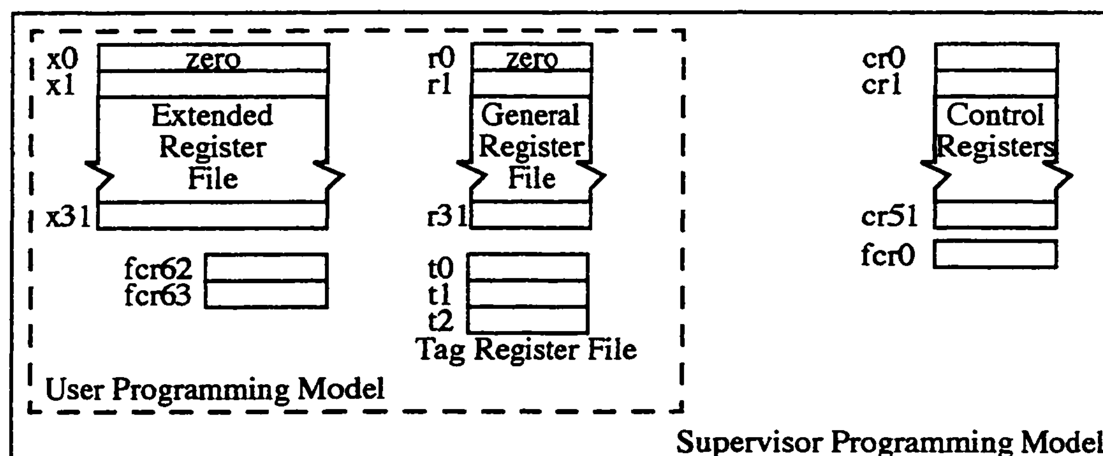


Figure 38 110L Programming Model

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

exactly the same for the 88110 and the 110L with the exception that the data cache and memory management unit control register will implement the *tag enable* bit required by LOTA. This addition will be discussed along with the data cache and memory management below.

As presented in Chapter 4, LOTA only involves the data access instructions and the data path to memory. Figure 39 presents the data access path from Figure 37 modified for the 110L. In the figure the addition to the 88110 design for implementation of LOTA has been highlighted by shading. The tag register file has been shown as part of the data unit to emphasis its closely linked operation. The data path between the tag register file and the internal data bus is only utilized for transferring tag registers to and from the general register file. Memory access

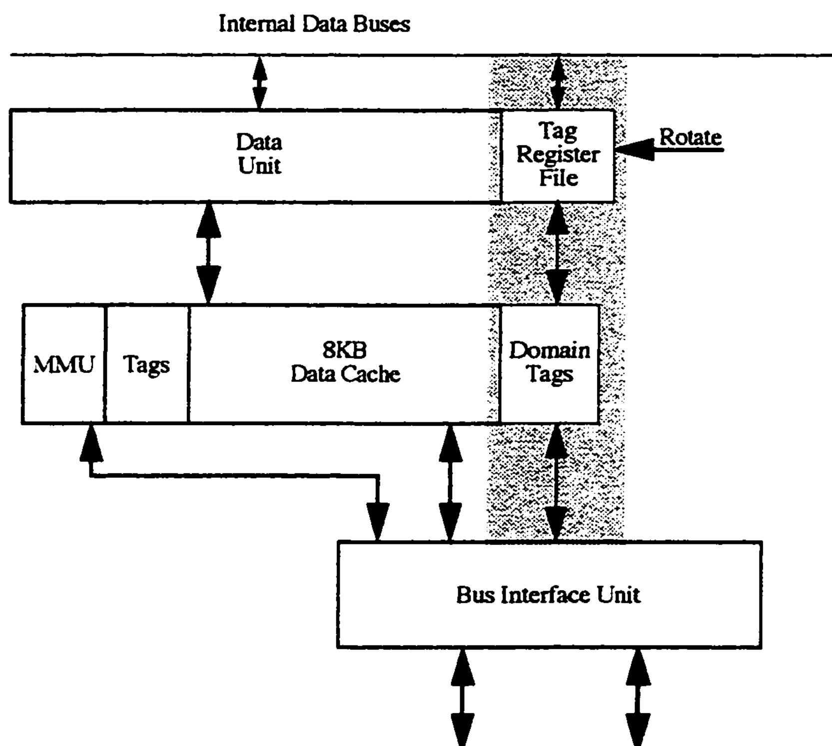


Figure 39 110L Data Access Path

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

instructions are managed by the data unit in the context of the domain specified by the tag registers as described in Chapter 4. The *Rotate* flag is passed from the instruction sequencing logic to indicate an *enter* instruction has been issued to change protection domains. The data cache has been expanded to include domain tags for each cache line along with the control logic to perform access checking. The bus interface unit along with its data path to the data cache have also been enhanced.

5.1.2.1 Instruction Set. All of the instructions defined for LOTA are implemented.

Load Tag Register. The *load tag register* instruction is implemented with the syntax *ldcr rD,tS*. Execution of this instruction will cause the tag register tS to be transferred to general register rD.

Store Tag Register. The *store tag register* instruction is implemented with the syntax *stcr rS, tD*. Execution of this instruction will cause the general register rS to be transferred to the tag register tD.

Enter Protection Domain. The *enter protection domain* instruction will be implemented with the syntax *enter rS*. rS contains the address of the subroutine. The return program counter is copied into r1. This instruction is a variation of the *jump to subroutine* instruction of the 88110 where the *Rotate* flag is sent to the tag registers to implement the domain change as defined by LOTA.

Return From Protection Domain. The *return from protection domain* instruction is implemented with the syntax *return*. The address used for the return is contained in r1. This instruction is a variation of the *jmp r1* instruction of the 88110 where the *Rotate* flag is sent to the tag registers to implement the domain change as defined by LOTA.

Load Register. The *load register* instruction is unchanged from the standard *ld*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

instruction of the 88110.

Store Register. The *store register* instruction is unchanged from the standard *st* instruction of the 88110.

Store Register with Tag Update. The *store register with tag update* instruction substitutes *stu* in the syntax of the *st* instruction to indicate that if the store operation is allowed the tag should be updated with the value in t0 as required by LOTA.

Update Tag. The *update tag* instruction substitutes *stt* in the syntax of the *st* instruction to indicate that if access is allowed to the specified location then the tag should be updated with the value in t0 as required by LOTA.

5.1.2.2 Instruction Unit. The instruction unit must also be enhanced for the 110L but its changes do not show up on a block diagram. There are three basic enhancements: knowledge of tag registers and domains, issue logic for the added instructions, and hazard information for instruction issue constraints. The instruction unit implements the *load tag register* and *store tag register* instructions directly, transferring the appropriate tag and general register values by gating the information over the internal buses. It also implements the *enter domain* and *return from domain* instructions. These are implemented as normal subroutine call and return sequences but accompanied by the *Rotate* control flag to indicate a protection domain change. All the other tag related instructions are issued to the data unit for execution.

The 88110 implements the instruction issue logic by implementing two issue slots. These slots, labeled issue slot 0 and issue slot 1, are filled with instructions from the program in the order contained in the program. The 88110 issues instructions in program order but allows for their completion out-of-order. It may issue up to two instructions per clock. To maintain program issue order the issue slots are filled in order. Issue slot 0 must be issued before, or at the same time as, issue slot 1. In some cases both issue slots cannot be issued simultaneously due to a conflict in the resources

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

needed by the respective instructions. These conflicts are referred to as issue constraints. The 110L contains all the same issue constraints as the 88110. In addition it contains the constraint that two tag related instructions cannot be issued simultaneously. Since the data unit already has the constraint that it can except a single instruction per clock cycle the 110L adds no additional constraints for tag instructions issued to the data unit. However, the enter *protection domain* and *return from protection domain* instructions also effect the tag registers. Therefore, the 110L has the additional issue constraint that these instructions cannot be issued in the same cycle as instructions to the data unit. This constraint is exposed to the compiler for optimization of instruction ordering.

5.1.2.3 Data Unit. The data unit in the 88110 is the most complex execution unit and performs all data memory access. There is an arithmetic unit on the input of the data unit in order to perform address calculation. Any information the data unit needs in order to perform address calculation, namely the value of a register or immediate value encoded in the instruction, must be available at the time of instruction issue. The data unit can accept one load or one store instruction per cycle. After immediate address calculation the instruction is inserted into one of two first-in-first-out (FIFO) queues. Load instructions enter the *load buffer* queue while store instructions enter the *store reservation station* queue. The FIFO organization of these queues imply that store instructions flow in program order with respect to other store instructions. The same is true of load instructions. Having two separate FIFO queues could allow load and store instructions to flow out-of-order with respect to each other. The 881110 allows only load instructions to flow ahead of stalled store instructions. Once an instruction flows to the end of the queue it enters a two stage pipeline to access the data cache.

The load buffer is a four deep FIFO and the store reservation station is a three deep FIFO as shown in Figure 40. The only difference between the 88110 and the 110L is the domain fields which are added to each instruction as required by LOTA. The store FIFO is implemented as a reservation station to indicate that the register value

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

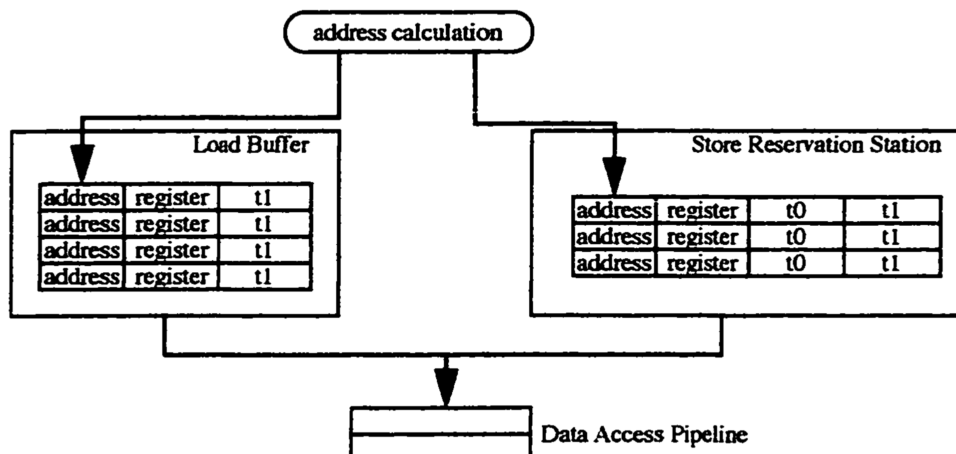


Figure 40 110L Data Unit

being saved does not have to be known at the time the instruction is issued. As long as the register value becomes available before the instruction reaches the bottom of the FIFO it will not stall the FIFO.

Data Hazards. The 88110 does not allow a load instruction to pass ahead of a store instruction that is referencing the same memory address. A memory conflict would occur otherwise. To perform this check the data unit compares the address field in store reservation station entries to the address in load buffer entries. Any match creates a barrier which the load instruction cannot pass. To simplify implementation in the 88110 the address match is performed only on the most significant 20 bits of the address. The lower 12 bits are ignored. These 12 bits represent 4096 bytes of storage and is also the size of a store page in the memory hierarchy. Therefore, load instructions cannot advance ahead of stalled store instructions when their address fall within the same page of memory.

The 110L must also assure that all instructions which change the domain field of a cache line remain in program order. Only the *store register with tag update* and *update tag* instructions effect the domain field in a cache line. Both of these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

instructions flow through the store reservation station queue in the data unit. This implies in the 110L a load instruction to a cache line must not be allowed to pass a *store register with tag update* or *update tag* instruction to the same cache line. However, cache lines never straddle page boundaries in the 88110 or 110L. Therefore the constraint on order of loads with respect to stores on a cache line is a subset of the constraint on order of loads with respect to stores within a page. Hence the 110L has no additional constraints to implement in the data unit.

5.1.2.4 Data Cache and Memory Management Unit.

The operation of the data cache and the data memory management unit (DMMU) are closely related and will be presented together here. An address in the 110L is 32 bits in length. Cache is organized as 128 sets of two cache line entries each (2-way set associative). Each cache line contains eight 32 bit memory words. In order to speed up the process, the DMMU translates the virtual address in parallel with set selection in the data cache. This is accomplished by splitting the address into parts to work on independently and in parallel as shown in Figure 41. Since memory is managed in terms of pages, each 4096 bytes in length, the DMMU only needs the 20 most significant bits of the address to translate from a virtual page address to a physical page address. Bits 5 through 11 are used to select one of the 128 sets in the data cache *in parallel* with the DMMU address translation. The translated address (the physical

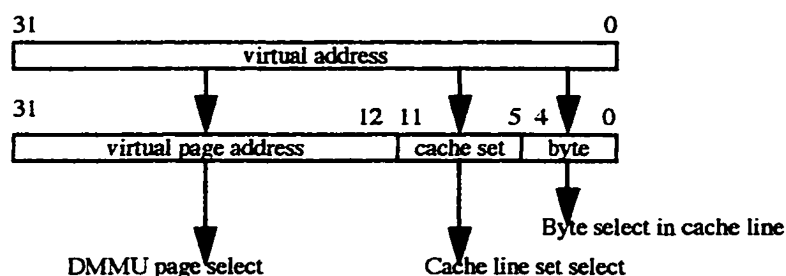


Figure 41 Virtual Address in the 110L

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

page address) is then used to determine if one of the entries of the selected set contains the target memory. The design of the cache is utilizing part of the virtual address to index the cache while physical addresses are used as tags. This is a common trick known as a virtually indexed and physically tagged cache structure [52].

The data cache operation, following set selection, is demonstrated in Figure 42. In the first step following set selection the address tag fields of both cache lines are compared in parallel to the translated address from the DMMU. During this same time period the domain fields are compared to the tag register values to determine if the access is allowed. In the next step control logic must determine if there was an address tag match and determine from which line. This step would also include the domain compare result from the same cache line. In the final step the control logic must use the remaining 5 bits of the original virtual address, along with the various control information, to multiplex the data to or from the correct location within the cache line. During this step any tag update operation must also be performed.

The LOTA domain check in the cache operates in parallel to functions already necessary. The result should be a cache which operates at the same speed. In the 110L the *tagging enable* bit will be implemented in the DMMU cache control register, shown in the supervisor programming model as cr41. This register has several undefined and reserved bits. The state of the tagging enable bit is distributed to all parts of the 110L which require knowledge of optional tagging features. This bit is set for

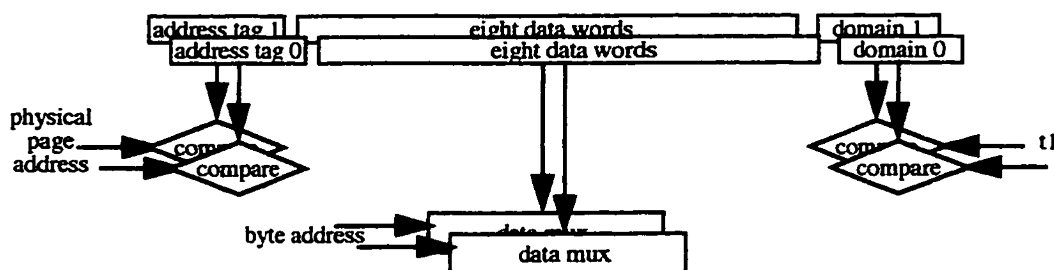


Figure 42 Cache Operation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

the process by the operating system on task switch.

5.1.2.5 Bus Interface Unit. The bus interface unit arbitrates between requests from the data cache and the instruction cache for access to the external bus. The external data bus is 64 bits while the address bus is 32 bits. Normal transfers between cache and memory is an entire cache line. Instruction cache transfers are unchanged when compared to the 88110. Since the data cache is organized as eight 32 bit words the 88110 requires four consecutive transfers on the external 64 bit bus to transfer the entire cache line. For a read the 88110 sends the address of the word in memory it needs and the memory responds with the four data words for the corresponding cache line. The 110L complicates this process by requiring an additional transfer of the tags. The read cache line process is demonstrated in Figure 43. In the figure the critical word indicates which word was requested by the program. In this case the critical word is in cache line *n* and is located at *n*+2. The 88110 transfers the critical word first followed by the next three memory locations counted with a modulus four counter. The 110L transfers the tag in a separate cycle and is first. In this manner the domain check can be performed before the critical word is forwarded to the processor. In this

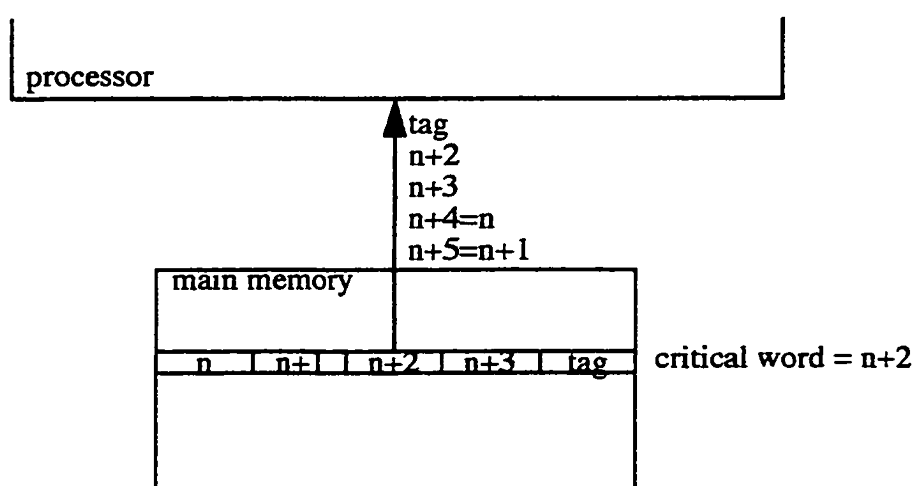


Figure 43 External Data Transfer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

implementation one extra cycle is added to the access of the critical word. Since the processor may be waiting for the critical word in a computation this may result in additional stall cycles for the processor. This should be minor as the 88110 has many features to allow continued processing in the face of data dependencies.

5.1.3 Cost Analysis

The costs of implementing the 110L as compared to the 88110 can be enumerated along the lines of information flow through the processor.

1. Instruction Unit - The instruction unit is enhanced only slightly. It must understand tag registers and signal when domain crossing instructions are executed. The actual domain crossing instructions are variations of normal subroutine calling instructions where the difference is limited to signalling the domain change.

2. Tag Registers - Three 32 bit tag registers have been added. Added logic must rotate these registers when a domain change is signaled.

3. Data Unit - The data unit has the load buffer entries expanded by 32 bits and the store reservation station queue entries expanded by 64 bits. This represents the addition of ten 32 bit words of storage capacity to the data unit. The queue operates without added complexity but the pipeline to the data cache must be aware of the added information flow.

4. Data Cache - The data cache entries are expanded by one 32 bit domain field. The data cache already adds three state bits and one 20 bit address tag to each cache line of eight words, for a total of 279 bits of storage. The 110L requires 311 bits per cache line representing 11.5 percent overhead in cache storage. In addition there are two 27 bit comparators added for domain comparisons along with the additional logic to update the domain field. The data path between the data unit and data cache has been expanded, for store operations, by 64 bits. However, a clever implementation would realize the two 32 bit values are needed at separate times and could possibly multiplex

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

the data over a single 32 bit path without effecting the cycle time.

5. Bus Interface Unit - The bus interface unit has additional logic to optionally perform four or five data transfers per cache line transfer. This will probably require 32 bits of additional temporary storage but only a small amount of logic since it does not need to interpret the tag bits.

5.2 Experiments

5.2.1 The Simulator - XSim

XSim is the Motorola instruction level simulator for the 88110 processor [53]. It attempts to simulate as accurately as possible the operation and timing of the processor. XSim is very useful to estimate performance of real programs on the 88110 in the absence of hardware [54]. XSim operates as a command interpreter and accepts basic commands to load a program, display memory or registers, map program memory, and to log statistics to a file.

XSim provides visibility of the 88110 at clock cycle resolution. The internal states which are visible include:

- cache - includes data, state and tag
- clock - current clock cycle since program start
- history buffer - the complete state of the history buffer
- map - the current memory map
- program memory
- registers - general, extended, and control
- target instruction cache - branch address, target address, two target instructions

In addition to the state of the processor, XSim also maintains some profile information on the program and statistics of processor operation. When profiling is on the simulator attempts to determine, from the symbol table information, what basic program block is executing and assign clock cycles to that symbol. A display of the profile information will display the address, symbol, number of times the first

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

instruction is executed (an attempt to determine the number of branches into a basic block), and the number of clocks in which the program counter was within a basic block.

Processor statistics are divided into the categories:

- Clock cycles
- Instructions issued
- Instructions issued per clock
- Issue stall distribution - 12 conditions which can stall instruction issue
- Instruction distribution - instruction distribution to the execution units
- Load/Store statistics
- Control flow statistics - branching statistics
- System calls - counts of operating system calls

These statistics provide valuable information for chip design verification and compiler optimization.

Cache Effects. XSim models the cache completely and statistics are gathered for hit rates. Although the cache is simulated correctly, the simulated environment does not take into account operating system calls or effects of operating system timesharing. In effect the cache is private to the program and does not compete with the operating system. This will tend to give hit rates higher than would be expected in real systems.

Operating System Calls. XSim provides for a limited number of operating system calls. These calls are performed by software traps which, in a real system, would take the processor into the operating system to perform the operation. These traps are detected by XSim and optionally emulated. When emulation is enabled, XSim halts the simulation and then performs the system call to the native operating system in which XSim is running. The results are assembled such that, when the simulation resumes, the operating system call appears to take zero time. The simulator model is limited to correct timing for user mode instructions only. The operating system calls supported

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

are:

- brk - change the data segment space allocation
- close - file close
- exit - program and process exit
- ioctl - input/output control
- open - file open
- read - file read
- time - get system time
- times - process times
- write - file write
- stat - file status
- xstat - SVR4 file status
- fstat - file status
- fxstat - SVR4 file status
- lseek - position offset for file read/write
- sysconf - system configuration

110L Simulation. XSim cannot simulate the LOTA enhancements to the 88110. Altering XSim to simulate the 110L would require a LOTA enhanced compiler to be useful, an unnecessary and complicated addition to these experiments. Instead the simulator is enhanced to report additional information about the machine and the executing program. The program under test is also modified. Together these modifications allow the performance of the 110L to be projected. The performance penalties can be extracted and categorized.

For the programs tested to operate properly, the following system call support was added to XSim:

- getpid - get process ID
- fcntl - file control

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

- access - determine accessibility of a file
- creat - create a new file or rewrite an existing one
- unlink - remove links to files and directories
- lstat - file status

The statistics gathering of XSim was enhanced to include these categories:

- stack register updates
- stack bytes allocated
- stack allocated in terms of cache lines
- stack bytes deallocated
- stack deallocated in terms of cache lines
- maximum stack depth in bytes
- maximum stack depth in cache lines
- number of calls to malloc
- bytes allocated by malloc
- heap allocated by malloc in cache lines
- number of calls to free
- number of subroutine calls
- number of return from subroutine jumps
- number of leaf nodes - routines which do not make additional subroutine calls

5.2.2 Measurements

The 110L will pay performance penalties in three ways: added instruction processing to manage tags, added instructions to manage domains, and added time to transfer cache lines between memory and cache. The added tag management is needed to cross protection domains, free stack space, and to claim and free heap space.

To estimate these penalties several C++ programs are compiled and simulated. The simulation environment has severe constraints as to what type of program may be simulated. First, all system calls must be trapped and emulated with direct calls to the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

operating system where the simulation is taking place. Second, programs must be statically linked. Statically linked programs have all function pointers resolved so any library function needed is included with the program. Ordinarily, programs are started on Unix systems with the *exec* system call which loads the program and starts its environment. Dynamically linked programs require *exec* to cooperate with the dynamic linker to set the memory map up for the program. XSim performs the function of *exec* in the simulated environment and must act as the master of the memory space. XSim cannot handle any situation where the memory map of a program must be altered dynamically for proper operation. This does not include heap space memory allocation. The slow simulation speed, about 10,000 instructions per second, make many programs impractical for simulation.

Simulation Parameters. XSim has a few parameters which need to be set. Unix system call emulation is turned on as well as profiling and statistics gathering. Data and instruction caches are turned on. The standard input and standard output are set to use a file when needed and input files are initialized before simulation. Command line arguments are set for programs which require them. Each program tested is given a memory space for use as stack and register r31, the stack pointer, is initialized to the bottom of this space. All of these parameters are set for each program and remain the same for all simulation runs for that program.

Memory Simulation. When a cache miss occurs XSim simulates the time to fill the cache from memory. XSim has two values for simulating the number of clock cycles to transfer a cache line. The first value is the number of wait cycles to access the critical word (first word) of the transfer. The second value is the number of wait cycles for all additional words of the transfer. Since there are four transfers for a cache line, setting XSim to (3,1) cycles would indicate three cycles for the critical word and one cycle for each of the next three words of the cache line. This is referred to as a 3-1-1-1 burst transfer. The 88110 based MVME197 [55] computer board advertises a 3-1-1-1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

sustained transfer rate so this value is used for the standard 88110 simulations.

The 110L must transfer the tag field in addition to the words of a cache line. The burst transfer is extended to five word transfers with 3-1-1-1-1 burst rate. Since XSim is unaware of tags a 4-1-1-1 transfer will be used to emulate it. The first transfer in the 110L is the tag followed by the critical word. Since the critical word is delayed by one extra cycle while the tag is transferred ahead of it, the four cycle delay will cause the same effect in the simulated performance. The 88110 forwards the critical word to where it is needed without waiting until the cache line has completely filled. The 110L therefore has no choice but to transfer the tag first so that it can perform the domain check for LOTA before the data is used.

5.2.2.1 Experimental Runs and Definitions. It is clear that XSim must be run with two different operational settings. In the first a 3-1-1-1 cache burst transfer is used to determine the application base performance characteristics. In the second a 4-1-1-1 cache burst transfer is used to emulate the penalty for transferring the tag bits prior to the critical word. XSim has been modified to determine the number of subroutine calls and leaf subroutines encountered. It also determines the amount of stack space used. However, XSim cannot reliably determine the amount of heap space allocated without help from the software. C++ software can allocate heap space through the traditional library routine *malloc*, or it may use the C++ *new* function. The memory allocation function may also be implemented as part of the program. For XSim to correctly determine the amount of memory allocated each program was examined and made certain that only one memory allocation function is utilized. To facilitate this procedure new implementations of memory allocations functions were developed.

new. In C++ the function *new* is used to allocate memory and assure it is initialized. The compiler or library may implement *new* in any fashion it chooses but the language allows for the user to override the default behavior by supplying its own implementation. Each program was instrumented with the *new* implementation shown

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

in Figure 44. The function *delete* is a companion to *new* and allows for orderly

```
void* operator new(size_t s)
{
        return(malloc(s));
}

void operator delete(void* prt)
{
        free(prt);
}
```

Figure 44 Operator *new* and *delete*

destruction of objects. In C++ terminology these definitions overload the global operator *new* and *delete*. In the implementation these routines are linked into the program before the C++ library to assure they are called instead of the default implementations. These routines are extremely simple implementations which have only one goal, to assure that *malloc* is used in the implementation of *new*.

malloc. The system supplied *malloc* and associated routines are also implemented with special versions for application in this research. *malloc* is the memory allocation routine and it operates by using the *sbrk* operating system call to expand the memory space of a program. This expanded space is then used by *malloc* to allocate dynamic space to the program. This space is referred to as program heap. There are several implementations of memory allocation routines available. Each have their own philosophy behind their implementation and all claim to be more efficient than the others. In reality each operates best under certain circumstances and all have cases where they perform poorly. A few of these versions were tried for use in this research. However, they tended to be highly optimized and very sensitive to modifications. The design decisions for an implementation here are quit different and made building a memory allocation function from scratch most practical.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

The main difference in the requirements for implementing *malloc* is that memory should be allocated in terms of cache lines and be aligned on cache line boundaries. This would be required of any implementation of LOTA. In fact, all memory allocation routines must deal with alignment and the value used for alignment is not very significant. Alignment in this case is performed as shown in Figure 45 where *nbytes* represents the request in bytes. Most of the remaining implementation

```
#define ALIGN 32

    if( nbytes & (ALIGN-1) )
       nbytes += ALIGN - nbytes % ALIGN;
```

Figure 45 Alignment in *malloc*

details are not of much interest. *malloc* requests one megabyte at a time from *sbrk* for heap space to satisfy small requests and uses *sbrk* directly for large requests. One additional cache line is allocated to record the size of the request to assist in statistics gathering.

There is one detail which is of interest, tag overhead is optionally emulated. LOTA requires tags to be initialized for each cache line allocated and again when these lines are returned to the heap. Figure 46 shows the code used to emulate tag initialization. *ptmp* is the pointer *malloc* returns to the calling function and *nbytes*

```
#ifdef TAGS
    for(i=0; i<nbytes; i+=ALIGN) *(ptmp+i) = 0;
#endif
```

Figure 46 Emulation of tag initialization

contains the requested number of bytes. Emulation of tag overhead is accomplished by writing one location within each line of the allocated buffer. This exactly matches the LOTA requirement that one *update tag* instruction be used to change the tag value of each line allocated. It also emulates exactly the effect on the cache for accessing each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

line allocated. This effect could range from beneficial pre-loading of small allocations to cache busting (flushing useful data) worse case behavior. The *#ifdef TAGS* statement allows this emulation to be optionally compiled into *malloc* and requires two compilations and hence multiple experimental runs.

free. The function *free* is the companion to *malloc* and returns space no longer needed back to the heap. *free* is called with just one parameter, the pointer to the buffer to return. The implementation of *free* discovers how large the buffer is by extracting the *nbytes* value stored in the extra cache line *malloc* allocated and initialized. *free* also contains the identical optional emulation of tag initialization overhead as *malloc*. The deallocation policy of *free* is to simply throw the buffer away. If a program has extremely large memory allocation needs this policy of ignoring deallocation of memory will cause a failure. Otherwise it is most efficient.

The functions *realloc* and *calloc* are also implemented to satisfy program link requirements. *realloc* simply frees one buffer and allocates a new one in its place. A copy of the data from the old buffer to the new one is also required. *calloc* is the same as *malloc* but clears all the memory. Both of these functions were implemented using *malloc*.

Experimental Runs. Each program is linked with the two versions of *new* and *malloc* as defined above. In addition, C++ has the ability to implement a procedure in-line without making a subroutine call. Each program is compiled an additional time to turn off in-line expansion of routines. This is necessary to discover the number of protection domain crossings. The experimental runs can now be enumerated.

1. normal run
2. normal run with tag transfers
3. in-line expansion disabled with tag transfers
4. heap emulation of tag initialization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

These enumeration values will be used as subscripts to identify experimental data.

Definitions. XSim reports many statistics and a large amount of profiling information. Much of this information is used to analyze results. Only several of these values are needed to quantify performance penalties for LOTA. These values are given definitions to assist in the following discussions.

- C - clock cycles for the execution of the program
- D - number of protection domain crossings
- F - heap space deallocated in cache lines
- J - number of subroutine calls
- L - leaf nodes, subroutines which do not call other subroutines
- M - heap space allocated in cache lines
- S - stack space allocated in cache lines

When used with subscripts these variables imply which experimental run they represent. For example, C_1 indicates the clock cycles to execute a program in experimental run one.The symbol Δ will be used to denote the difference function. For example, $\Delta(C_2,C_1)$ indicates the additional clock cycles experiment two required compared to experiment one for a program. This may be extended to additional variables.

5.2.2.2 Domain Crossing. To calculate the penalty to cross domains it will be necessary to estimate the penalty of a single domain cross and apply it to the information extracted with XSim. A subroutine can be divided into three parts, namely the preamble, body, and postscript. LOTA requires that the preamble of routines which make additional calls store the old domain value in the stack. The postscript of such routines must reverse the procedure. Saving the domain register requires one register-to-register transfer and one register-to-memory store. Restoring the tag register requires one memory-to-register load and one register-to-register operation. The preamble and postscript are performed once. The body of the routine must transfer a new domain value into a tag register for each new domain it calls.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Figure 47 represents code which would need be added to each routine which makes calls to other routines in different protection domains. The pseudo code on the

```
function() {                        function() {
                                        <preamble>
    <preamble>                          ldcr r8,t2
                                        st r8,r31,56


                                        stcr t0,r2,r2
    <body>                              <body>


                                        ld r8,r31,56
                                        stcr r8,t2
    <postscript>                        <postscript>
}                                   }
```

Figure 47 Representative Domain Crossing Overhead

left side represents the subroutine divided into its parts. The right side indicates the added instructions required for LOTA in the 110L. The preamble is expanded to transfer the old domain in t2 into a general register and then save it to the stack. The postscript recovers it from the stack and replaces it back into t2. The body is expanded by transferring the C++ implicit variable *this* to t0 to represent the domain to be called.

A program was written to take measurements in XSim to estimate the penalties involved in crossing a domain. For this purpose the object shown in Figure 48 was designed. This code is only a fragment of code used in a program. The compiler was used to generate assembly language for the *fun1* function and this assembly code was expanded to include representative instructions required in the 110L. In this object *b* is a pointer to an object of type *Square* and represents the new domain to be called.

The assembly language implementation of *fun1* returned by the compiler is shown in Figure 49. In this code sequence the lines in bold have be added to emulate the LOTA required domain management of tags. When executed on XSim with and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

```
class Atype {
        int     i;
        Square *b;
   public:
        Atype::Atype();
        int funl(int);
};
int Atype::funl(int i) {
        int c = b->square(i);
        return c/2;
}
```

Figure 48 Example class to determine domain penalty

```
;preamble
   subu  r31,r31,64
   st    r1,r31,60
   or    r8,r1,r1
   st    r8,r31,56
;body
   or    r7,r0,r2
   or    r6,r0,r3
   ld    r4,r7,4
   or    r2,r0,r4
   or    r3,r0,r6
   or    r8,r3,r3
   bsr   _square__6SquareFi      ;domain cross
   or    r5,r0,r2
   extu  r2,r5,0<1>
   bb0   31,r5,@L9
   subu  r2,r0,r5
   extu  r2,r2,0<1>
   subu  r2,r0,r2
;postscript
   ld    r8,r31,56
   or    r8,r8,r8
   ld    r1,r31,60
   addu  r31,r31,64
   jmp   r1
```

Figure 49 Assembly sequence for domain crossing

without the tag code sequences it was determined three additional clock cycles were required. This was true if the code was executed once or over many loops. On close

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

inspection, the added assembly language statements are simple register-to-register operations, or to memory very likely to be cached. They are also independent and easily scheduled for execution by the processor.

An immediate upper bound to the expected penalty for 110L can be calculated under the pessimistic assumption that every subroutine call requires the full domain change penalty. This value is $3 \bullet J$. There are two reasons why this value is pessimistic. The first is that not every subroutine call requires a domain change. Intra-object calls, for example, do not require a domain change. With the available data from XSim there is no method to determine which calls require domain changes and which do not. The second reason this is a pessimistic calculation is that routines which call several other routines only pay the preamble and postscript penalties once. Only the body penalty needs to be paid on every domain change. From XSim the number of subroutine calls is known and the number of leaf routines is known. The problem is to calculate how many preamble and postscript penalties this represents.

Figure 50 represents a call graph, a tree which contains a node for each subroutine and an edge for each subroutine call.The solid nodes are leaf nodes, nodes which make no subroutine calls themselves. The information provided for this graph

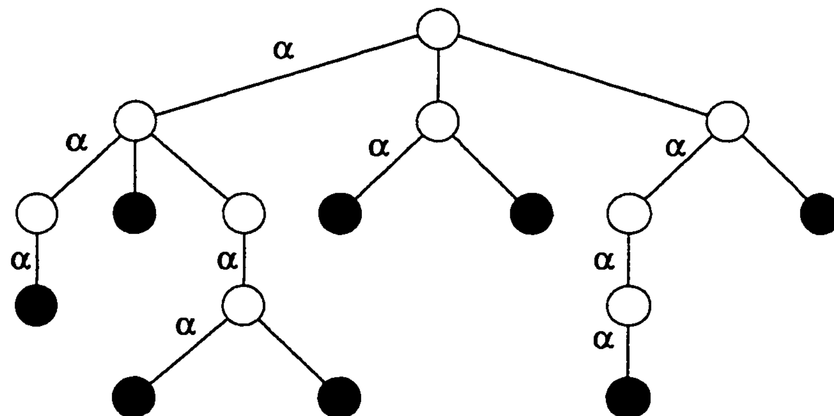


Figure 50 Call Graph

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

by XSim is the number of leaf nodes and the number of edges. All nodes which are not leaf nodes are called internal nodes. Notice, it is the internal nodes which are the only routines which must pay the penalty for saving and restoring domain values across calls. In Figure 50 the symbol α represents the number of preamble penalties which must be paid. For each preamble there is a corresponding postscript penalty denoted γ.

It is easy to see from the graph that the total number of nodes is the sum of the edges plus one. The number of nodes can be represented by the number of internal nodes plus the number of leaf nodes. Use A to denote the total number of internal nodes which in turn also represents the total number of preamble and the total number of postscript penalties. Equation 3 is the result.

$$A + L = J + 1 \qquad \text{(EQ 3)}$$

$$A = J + 1 - L \qquad \text{(EQ 4)}$$

The *one* in Equation 4 represents the root node of the graph. In a program it is not necessary to save the calling domain value in the root node. Even if it is saved by convention, the effect is small and will be dropped here. The total penalty a program is expected to incur due to domain crossings needs one more piece of information before it can be stated. It has been experimentally measured that a routine will expect to pay three cycles for the preamble penalty, the postscript penalty, and making a single call. It will be assumed that the penalty is equally distributed. This implies one cycle for each preamble, each postscript, and each call. Although this is an estimate, it is noted that the call penalty in the body of a routine is a simple register-to-register copy. Assigning it an equal part of the penalty is probably a worse case scenario. In addition, it is this penalty which is paid on *every* call. The preamble and postscript penalties are paid only once. The resulting total penalty, T, can now be stated in words as the sum of preamble penalties, postscript penalties, and domain call penalties.

$$T = (J - L) + (J - L) + J \qquad \text{(EQ 5)}$$

$$T = 3J - 2L \qquad \text{(EQ 6)}$$

In terms of information flow the number of instructions which are required in a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

program is two for every preamble, two for every postscript, and one for each domain. With the same estimation that every subroutine call is a domain change, then the added information flow of instructions can be represented by Equation 7.

$$instructions = 2(J-L) \bullet 2 + J \quad \text{(EQ 7)}$$

$$instructions = 5J - 4L \quad \text{(EQ 8)}$$

The information flow in terms of data is shown in Equation 9.

$$data = 2J - 2L \quad \text{(EQ 9)}$$

5.2.2.3 Stack Management. Stack management has simple rules. Space must be claimed as its used and may be claimed for another object. The compiler is assumed sophisticated enough to make claiming space free in terms of performance since it can use the *store with update* instruction to store a value it needs to store anyhow while claiming the space at the same time. During free operations the compiler must schedule additional *update tag* instructions to return the stack to a free state. XSim has been modified to report the amount of stack space returned in terms of cache lines. For each of these cache lines an *update tag* instruction must set the tag to globally accessible. It will be estimated that one clock cycle is needed for each of these instructions. This estimation will be nearly correct as stack normally holds recent data and is most likely to be cached. The cache penalty is therefore simply stated as S, the number of cache lines of stack set free. In terms of information flow there will be S added instructions and S additional data transfers.

5.2.2.4 Heap Management. Heap management is very much like stack management. However, heap space memory must have its tags initialized when allocated and deallocated. XSim has been modified to keep track of the number of bytes and lines the *malloc* function has given to the program under simulation. However, the *malloc* function has been implemented in a way to emulate tag initialization. Therefore the estimated performance penalty will be calculated from the measurements as shown in Equation 10.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

$$H = \Delta(C_4, C_2) \quad \text{(EQ 10)}$$

In terms of information flow, estimating that all lines allocated are also deallocated, there will be *2M* additional instructions. Since all these instructions also cause a data memory cycle there will also be *2M* data transfers.

5.2.2.5 In-line Functions. Some routines are not called in the traditional sense. Instead of a subroutine call and return, the function is expanded in-line. These functions still represent domain crossings. In the spirit of a subroutine without a call-return pair as with in-line expansion, the 110L will allow the compiler to directly change the current domain by writing the current domain register t1. In effect the penalty could be calculated by expanding the equations derived for domain crossing above. Given T was used to denote the domain crossing penalty, the added penalty for crossing domains due to in-line expansion can be estimated as $\Delta(T_3, T_2)$.

Cfront allows programs to be compiled without in-line expansion. When this option is chosen the compiler guarantees functions will not be expanded in-line. Experimental run three collects data on all programs compiled with in-line expansion turned off.

5.2.3 Software Tested

All programs tested are C++ programs and are compiled using the Cfront C++ translator. Each program was compiled in two fashions. The first was a normal compile. The second was with in-line expansion turned off. These two versions of the program were linked with the experimental implementation of *new* and *malloc*. A third program version was created by linking with a version of *malloc* which emulates the overhead of initializing tags. Therefore three versions of each program exist. Each program will use a suffix of *N* to denote normal compilation, *I* to denote in-line expansion turned off, and *C* to denote the cache line tag emulation version of *malloc*.

Bill of Materials. The Motorola Computer Group uses a C++ software set to generate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

and check the bill of materials for software products [56, 57]. The set includes three programs, *listbom* for listing the bill of materials, *genbom* for generating a list of materials, and *chkbom* for checking a previously generated bill of materials. The input to *chkbom* and *genbom* programs is a directory structure containing the files to be included in a software product, *chkbom* also uses as input the file name of the previously generated bill of materials. The output of genbom is the file containing the bill of materials, formatted for release to the factory.

groff. The GNU text formatting package groff contains six programs to format text into a postscript file. This software was discussed previously in Section 2.3.3.3 on page 21 for the research into software defects.

Class Library for High Energy Physics. The Class Library for High Energy Physics (CLHEP) is, as its name implies, a class library for applications in high energy physics [58]. Three test programs supplied with the distribution are used for this research. These include the *MatrixD* and *MatrixF* tests, and the *Random* test. The *MatrixD* and *MatrixF* test utilize some list classes provided in the library and test several different matrix operations. The major differences between *MatrixD* and *MatrixF* are the class definitions in terms of standard precision floating point and double precision floating point as well as the corresponding different library support. Random tests the random class support and utilizes a few different distribution algorithms. Several tests are performed on these distributions.

hsim. *hsim* is a simulator for the flexible modeling of a memory management unit [59]. It takes as input a configuration for the translation lookaside buffer to be modeled including the number of sets and the number of elements in the set. The simulator then takes an address trace and determines the effectiveness of the translation lookaside buffer and gathers other statistics relating to the utilization of the buffer entries.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

mpsdemo. *mpsdemo* is a demonstration program for the OSE library [60]. The basic component of OSE is the OTCLIB, a library of generic components including support for error handling, error message logging, error recovery, program debugging, and event driven systems. *mpsdemo* uses features of this library to implement a simulation of a mobile phone system.

z80sim. *z80sim* is a program to simulate the Z80 microprocessor [61, 62]. The software package was actually an entire TRS 80 computer simulator. However, the Z80 portion was removed for use in this research. The simulator was tested by writing a Z80 assembly language program to exercise all of the instructions in the instruction set at least once. This code was assembled into machine code and fed to the simulator as a file, much like code in a read only memory (ROM) for starting a computer from power on.

photon. FeynDiagram is a library of C++ routines which allow a user to draw high quality postscript Feynman diagrams by writing a C++ program to describe the diagram [63]. *photon* is an example of use of the FeynDiagram library. *photon* does not need input and generates a postscript file as output.

5.2.4 Example Session

Each XSim simulation was performed with the Unix command *time XSim <cmd > typescript*. The *time* function reports the time each simulation requires to execute. The *cmd* file contains the basic XSim commands to simulate the execution of a given program. The *cmd* file is shown in Figure 51 for *z80sim* experiment two. The file contains the commands to set the cache file wait cycles, load a program, and log statistics. The *Exec* command executes another file of XSim commands. The contents of this file are shown in Figure 52. This file contains the stack setup commands and enables the floating point unit in the 88110. It turns on Unix system call emulation, enables the caches, and sets the command line arguments the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

```
Set Wait 4 1
Load /home/88k/bin/trash80m
Exec xsim.prep
Modify Profile On
Run
Log Session On profile
Display Profile
Log Session Off
Log Session On tstat.sim
Display Statistics
Log Session Off
Quit
```

Figure 51 XSim Command File

```
Map efff0000,efffffff
Modify Register General r31
effffff0
.
Modify Register Control cr1
0
.
Set Unix On
Set Caches Real
Set Stderr error
Set Args trash80
Set Pager Off
```

Figure 52 XSim Preparation File

program under test would ordinarily expect.

The output of XSim is divided into the profiling information and the statistics. Some example statistics is shown in Figure 53. Scripts were developed to convert these statistics files into tab separated text files suitable for importing into a spreadsheet.

5.3 Results and Analysis

5.3.1 Tag Transfers

Experimental run one tests each program under normal compile and simulation parameters to establish a baseline of performance. Experimental run two is identical with the exception that the simulator is set to extend the burst cache fill time to emulate

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

```
Instructions issued:              &168680934
Instructions issued per clock (average): 1.009218

Instruction Distribution:
Integer unit 1 instructions:      &60896769  (36.1%)
Integer unit 2 instructions:      &7040142   ( 4.2%)
Load instructions:                &45227784  (26.8%)
Store instructions:               &25616181  (15.2%)

Subroutine calls:
   Number of jsr/bsr:                                        &6833472
   Number of return from subroutines:                        &6833468
   Number of number of leaf nodes:                           &4805653
```

Figure 53 Excerpt of XSim Statistics

tag transfers. The overhead for this burden can be stated as in Equation 11.

$$\textit{percent overhead} = \frac{(C_2 - C_1)}{C_1} \times 100 \qquad \text{(EQ 11)}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Figure 54 shows the results from the experimental data. The figure demonstrates how

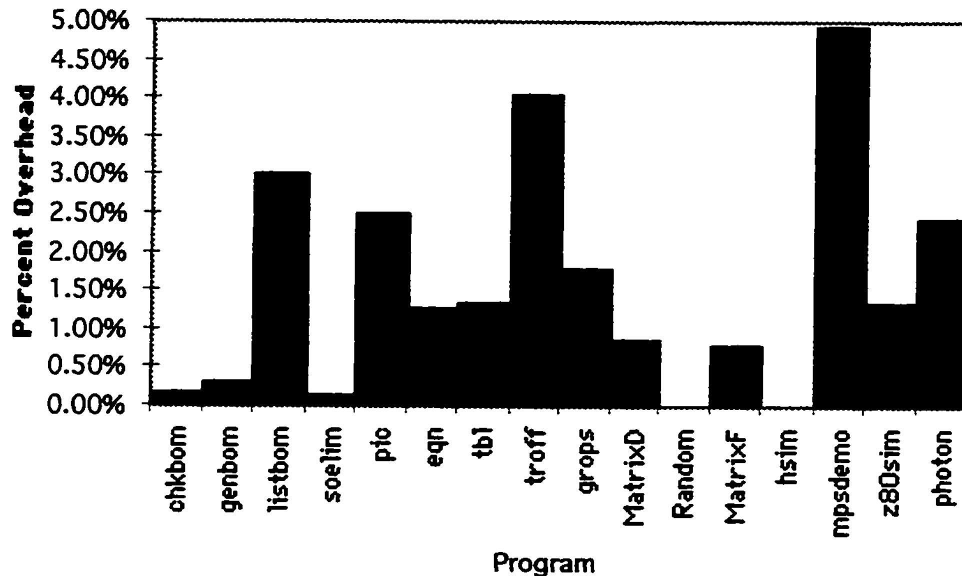


Figure 54 Tag Transfer Overhead

sensitive a high performance RISC processor is to external bus access. The overhead for a single extra cycle for access to the critical word of a cache line ranges from near zero to 4.9 percent. The cache effectiveness has a direct bearing on performance with respect to external bus speed. The more cache misses the processor must deal with the more often its progress will stall. Figures 55 and 56 show the effective cache miss rate for the programs. The same programs which have high miss rates also have greater sensitivity to bus performance. The two effects are not completely correlated. During the experimental research it was observed that a slower bus caused a slight improvement in performance in rare instances. The 88110 is aggressive in its attempt to continue making progress through out-of-order instruction completion and branch prediction. It sometimes makes mistakes which it must recover from by backing up. This makes the performance effects due to one parameter change difficult to predict.

The tag transfer performance penalty estimate here is higher than it should be due to the fact that the penalty is assigned for both data and instruction cache misses.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

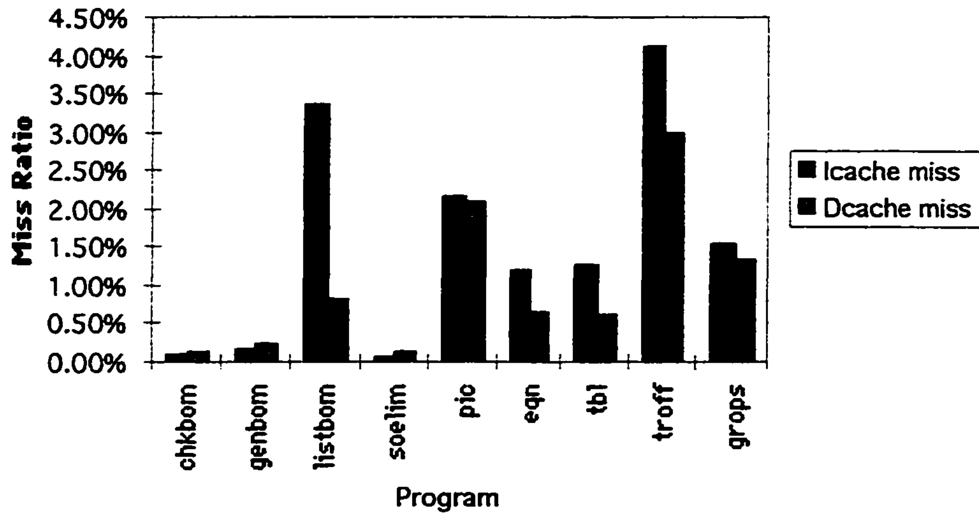


Figure 55 Cache Miss Ratios

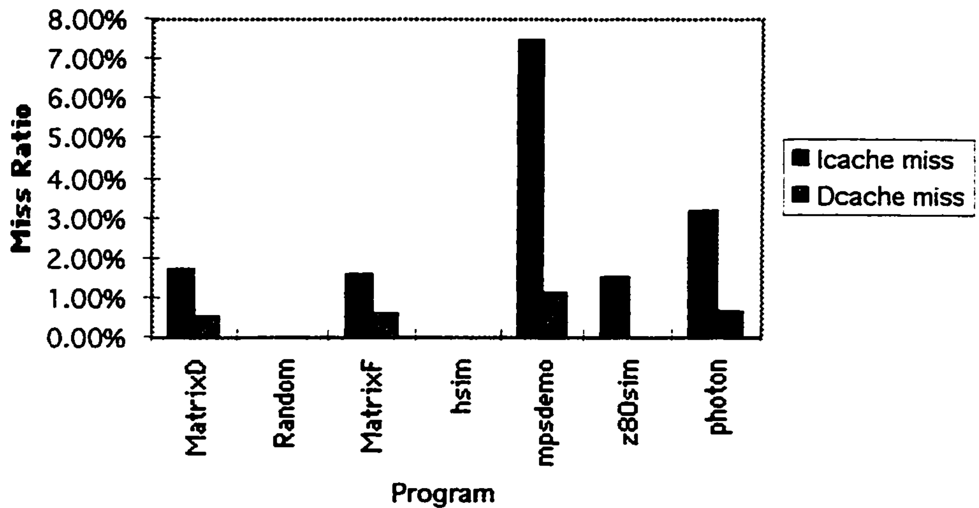


Figure 56 Cache Miss Ratio Continued

Only the data cache needs to carry the burden of an extra cycle to transfer tags. This will remain an area where the penalty estimate can be improved to favor better performance.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

5.3.2 Domain Crossing

The penalty for crossing domains is paid by each internal node of the call graph. The penalty estimate was given in Equation 6 to be $3J - 2L$. Figure 57 shows the domain penalty derived from experimental data. In all cases the penalty falls below

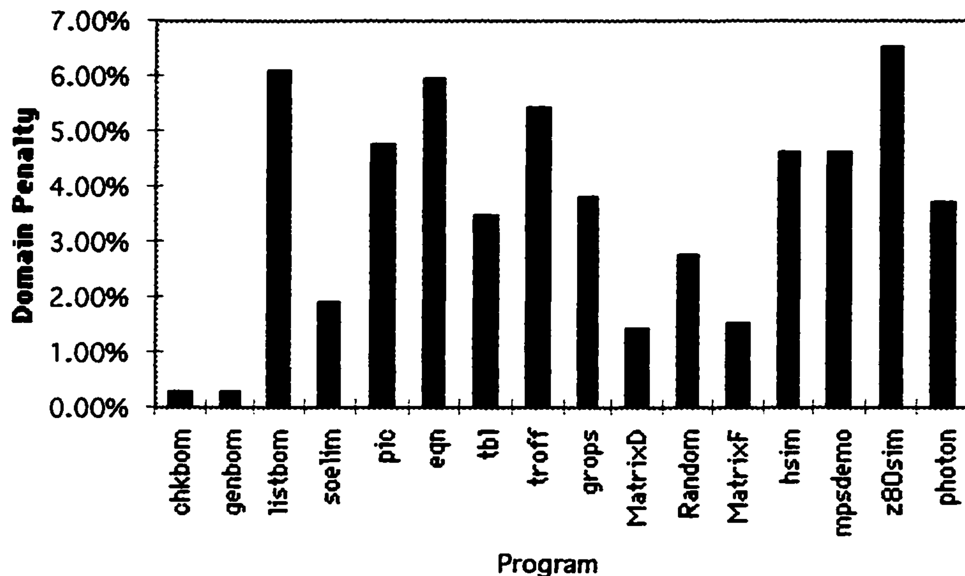


Figure 57 Domain Crossing

seven percent. This indicates LOTA holds promise with respect to domain crossing penalties. This is a very important part of the claims for LOTA and motivates continuing the research. Again, this estimate of the penalty is high. It assumes every subroutine call requires a protection domain change. An example where a protection domain change is not necessary is an intra-object call.

Consider the inherent added information flow for these domain changes. Equation 8 presented the additional instructions as $5J - 5L$ and Equation 9 presented the added data flow as $2J - 2L$. Figures 58 and 59 show these values derived from the experimental data. Each added instruction requires the processor to send an address for the instruction and receive back the instruction in the von Neumann tube between

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

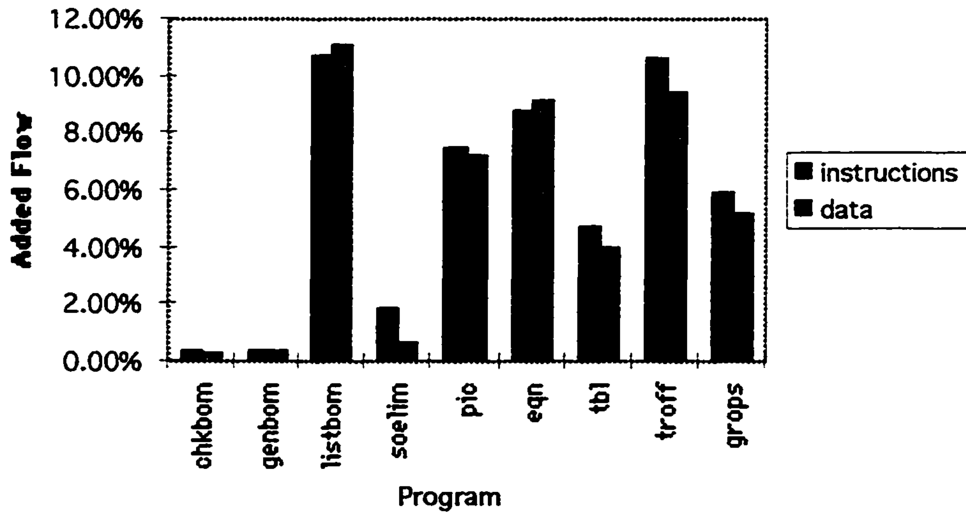


Figure 58 Information Flow

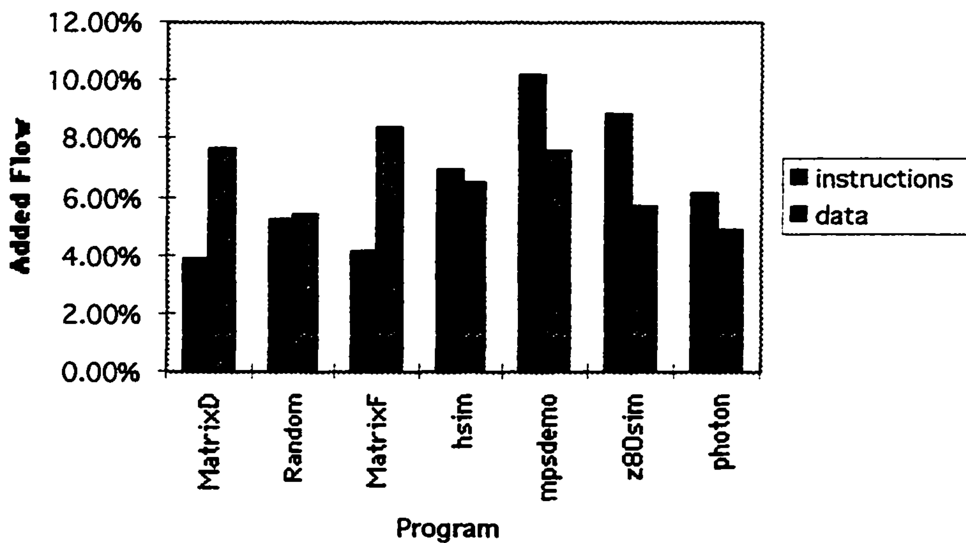


Figure 59 Information Flow Continued

processor and memory. The same is true for data. The difference in added flow between programs can be attributed to the program structure. The ratio of internal nodes to leaf nodes in the call graph will have a large effect on added information flow. Stated

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

another way, an increase in the number of times domain information must be saved and restored across calls causes a corresponding increase in required information flow.

5.3.3 Stack Space

A very closely related area to domain crossing is stack space management. Stack space is claimed as it is needed. When it is no longer needed the tags must be set to indicate the corresponding memory is available for reuse. The stack space operations are performed in direct proportion to the domain crossing. The performance of managing stack tags will be critical. The estimated penalty for managing tags in the stack is one *update tag* instruction for each cache line set free. Figure 60 shows the results obtained from the experimental data. For the majority of cases the overhead is

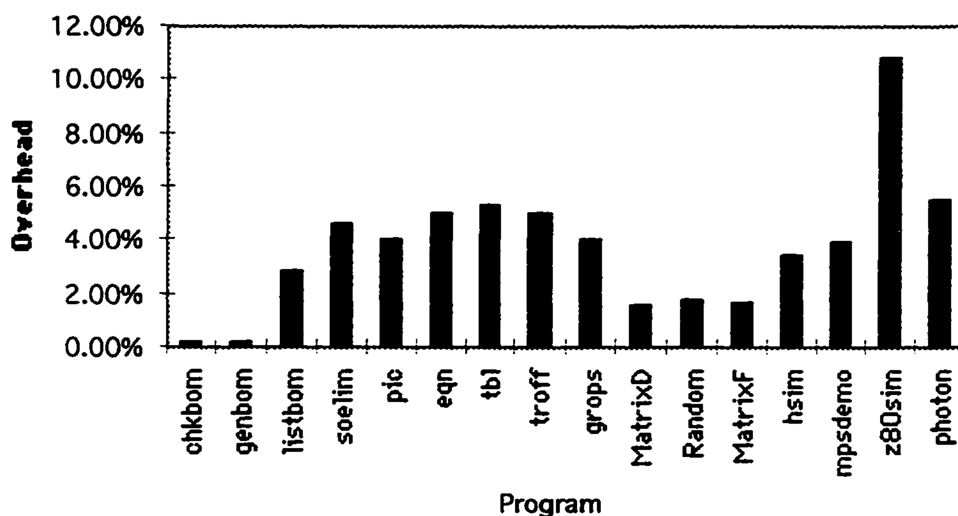


Figure 60 Stack Space Tag Management

less than six percent. The organization of the program may have a large effect on this overhead. Large structures or objects created on the stack will require much more overhead if these items are frequently constructed and destructed. This is not true if they are created on the stack of the root node and remain allocated until the program terminates. Any space the root node of a program claims will remain claimed for the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

life of the program.

The added information flow through the von Neumann tube is the number of lines deallocated in the stack space. The number is the same for added instructions as well as added data. Figures 61 and 62 demonstrate the extra flow.

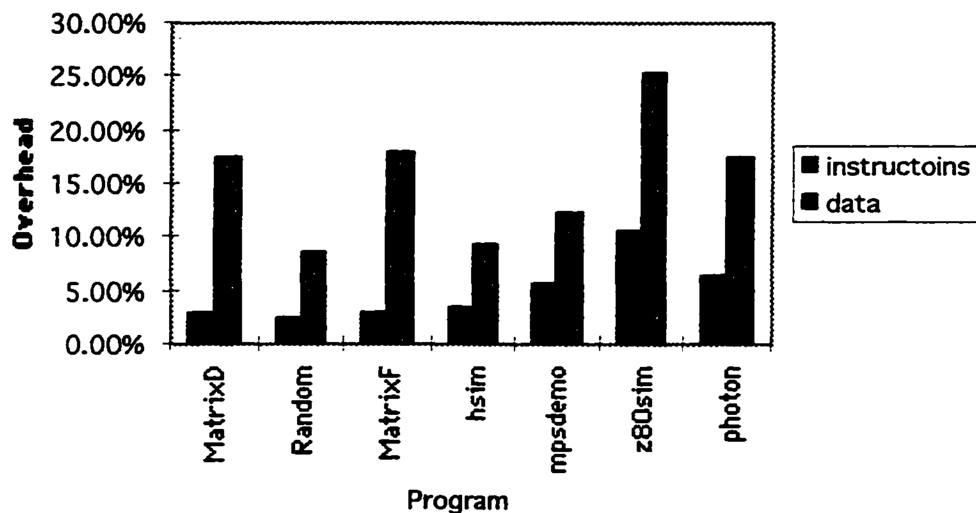


Figure 62 Stack Space Added Information Flow Continued

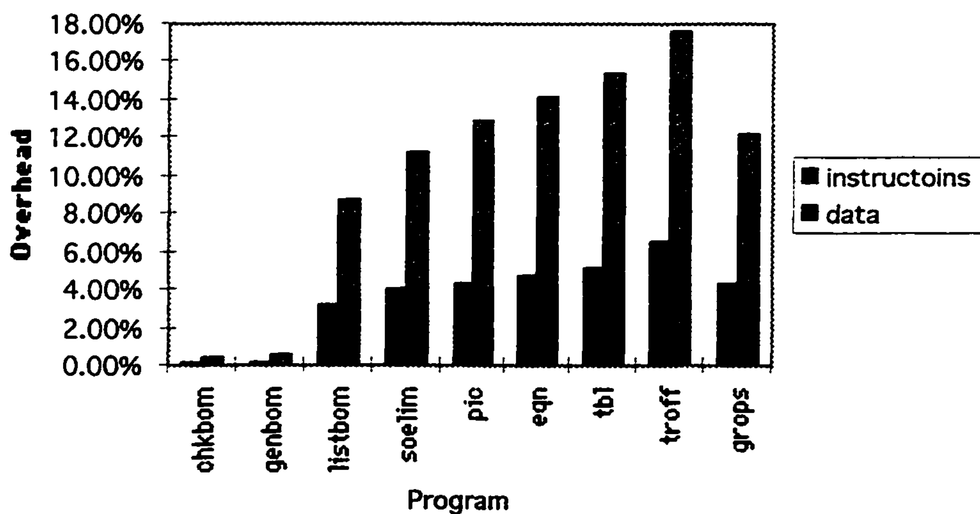


Figure 61 Stack Space Added Information Flow

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

5.3.4 Heap Space

LOTA added the extra requirement that data needs to be tagged with ownership at the time it is assigned to an object. In the heap space data must be allocated in terms of cache lines and tagged when allocated and have the tags cleared when deallocated. XSim provides the information necessary to determine the number of cache lines allocated. The programs have also been instrumented with a special version of memory allocation functions. Experimental run four uses programs linked with memory allocation routines which emulate the penalty of setting tag values for heap space. The estimated performance penalty was given in Equation 10 as $H = \Delta(C_4, C_2)$. Figure 63 presents the results obtained from experimental data. In all cases the overhead to set

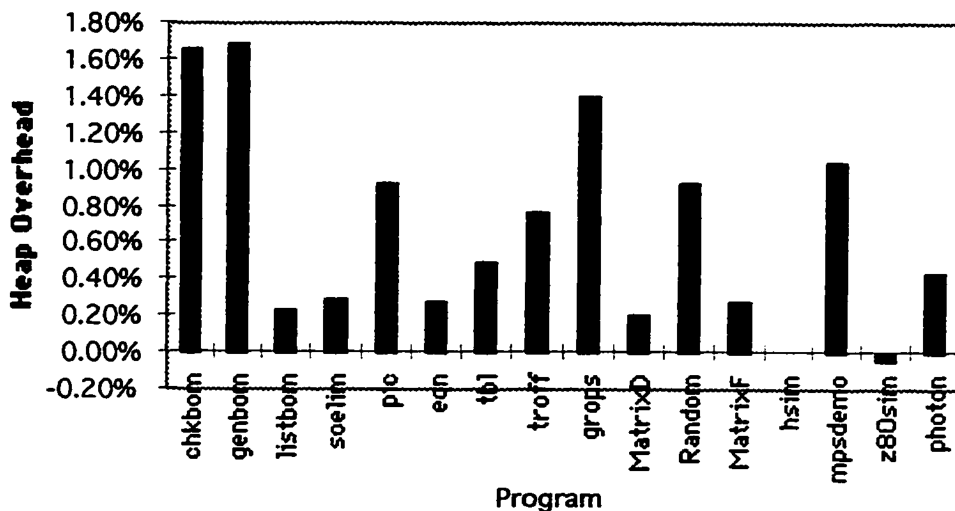


Figure 63 Heap Space Tag Management

and clear tags is very small. In fact, in one case initializing the tags pre-loaded the cache and caused performance to actually increase. Caution must be exercised interpreting this information. It appears that tag management is not a problem for heap memory. LOTA can be expected to have relative good performance when the size of heap space allocations is small. Small allocations would have small tag overhead which would be hidden in the normal processing requirements. If each allocation was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

large the tag overhead would become more apparent in two ways. First, the added instructions would naturally require more cycles of processor time. Second, as the allocation became large compared to total data cache available, the initialization of tags would tend to flush the cache requiring it to be refilled as necessary. To analyze why experimental data indicated a low overhead for heap space tag management the average memory allocation size is presented in Figure 64. As the figure shows, for the

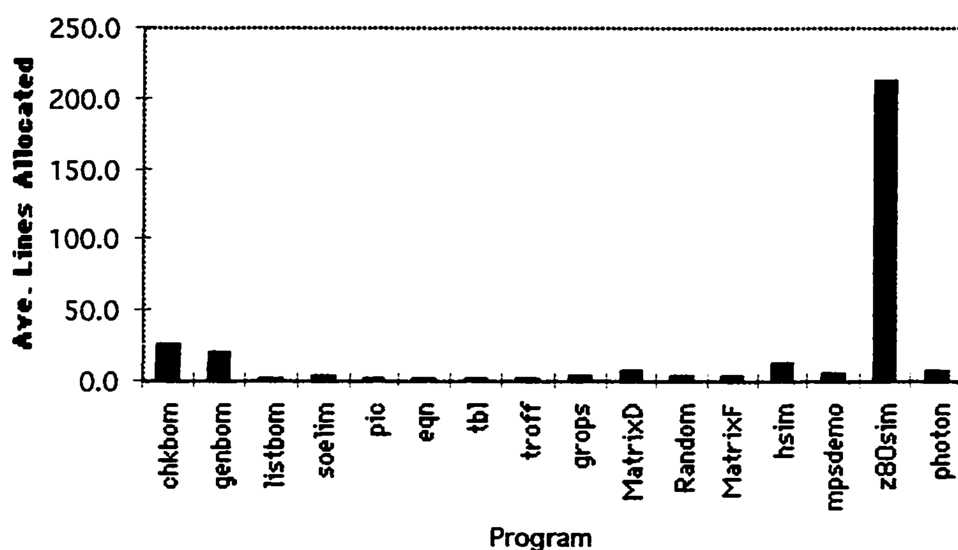


Figure 64 Average Heap Allocation

programs tested the heap space allocated per request is small in most cases. To understand the memory requirements of these programs it is helpful to compare the average allocation per request to the total number of bytes allocated. The total bytes allocated for each program during a simulation is shown in Figure 65. In general the heap space demand for this example program set is fairly small. This may not hold true for every program.

The added information flow due to the tag initialization is simply the sum of the cache lines allocated and deallocated. For this purpose it is assumed every line allocated is also deallocated. A single instruction is required for each line. All of these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

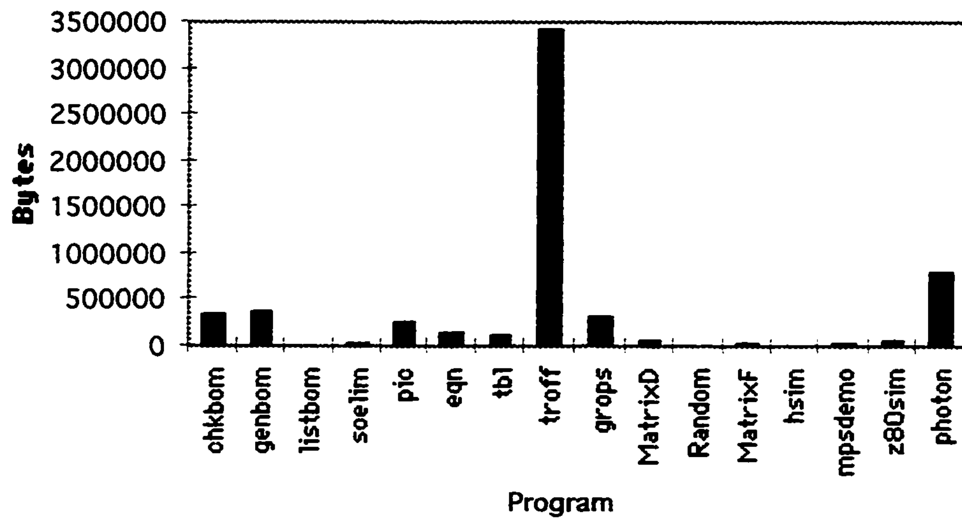


Figure 65 Total Heap Space Allocation

instructions are executed by the data unit and therefore represent a one-to-one correspondence in additional information flow due to instructions and data. This extra information flow is depicted in Figures 66 and 67.

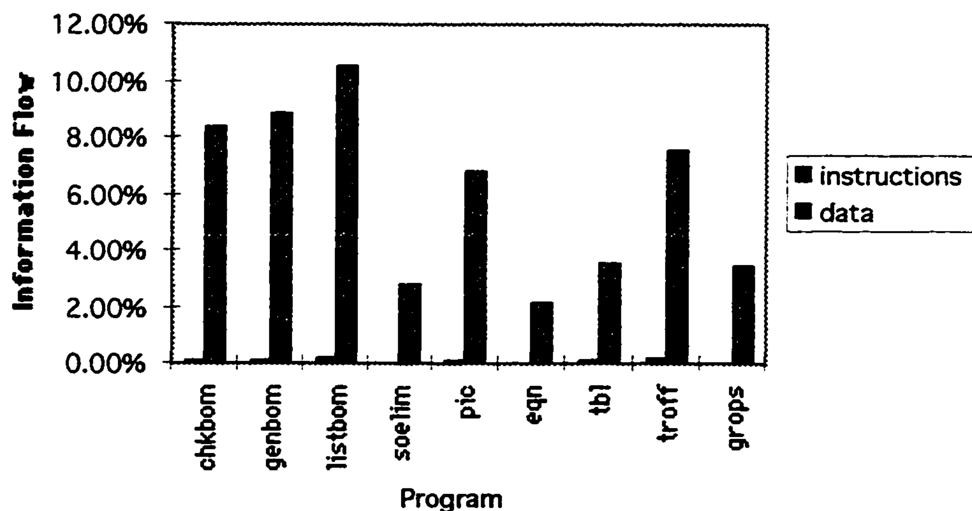


Figure 66 Heap Space Added Information Flow

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

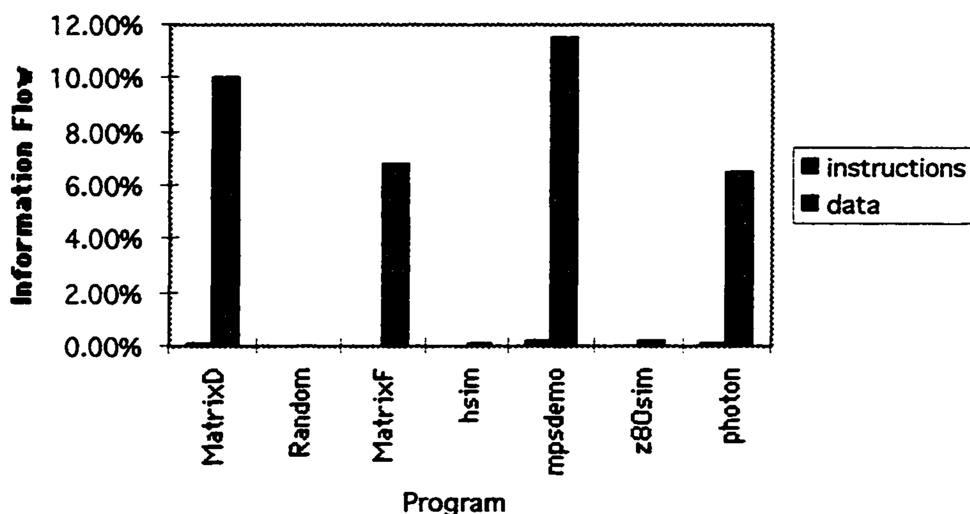


Figure 67 Heap Space Added Information Flow Continued

5.3.5 In-line Domain Crossing

One of the more difficult areas to deal with is that of in-line expansion of simple routines in C++. Instead of making a normal subroutine call and return, these functions are placed directly in-line. The in-line function expansion allows C++ to reclaim some of the performance penalty when compared to C programs while still maintaining the object-oriented program structure. LOTA can allow in-line expansion of domain crossing as well. To estimate the penalty for crossing domains in-line there must be a way to count them. The Cfront compiler has a compile time option to guarantee all routines are not in-lined. The extra subroutine calls and leaf nodes reported by XSim record the added domain changes. The domain crossing performance penalty derived from experimental data is shown in Figure 68. It is immediately obvious from the figure that the overhead in the *z80sim* program is out of line compared to the other programs. The other programs seem to be reasonable values which cause no serious performance concerns. Note, this experiment only made the extra domain crossings observable, the penalty estimate is based on the normal compilation with in-line expansion enabled. Consider the performance of the programs with in-line function

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

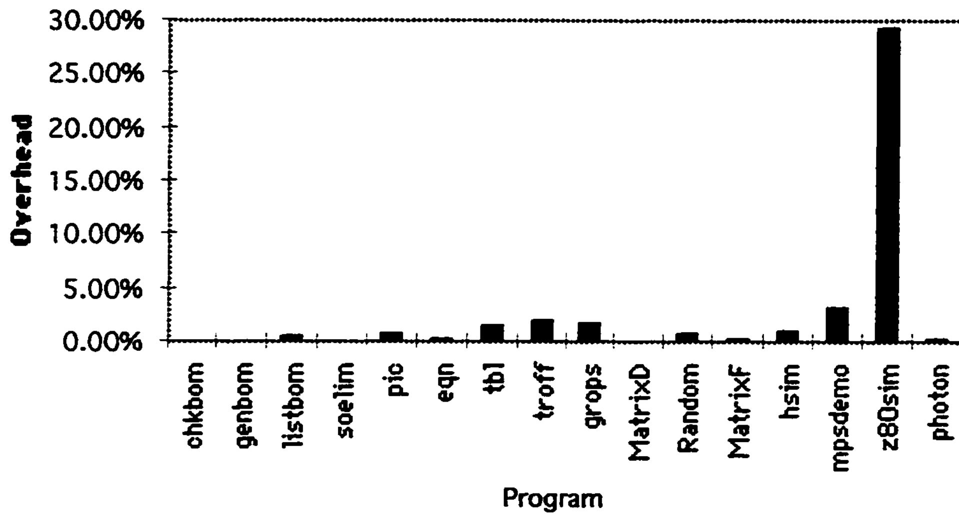


Figure 68 In-line Domain Crossing

turned off but no other penalty as shown in Figure 69. *z80sim* is obviously very

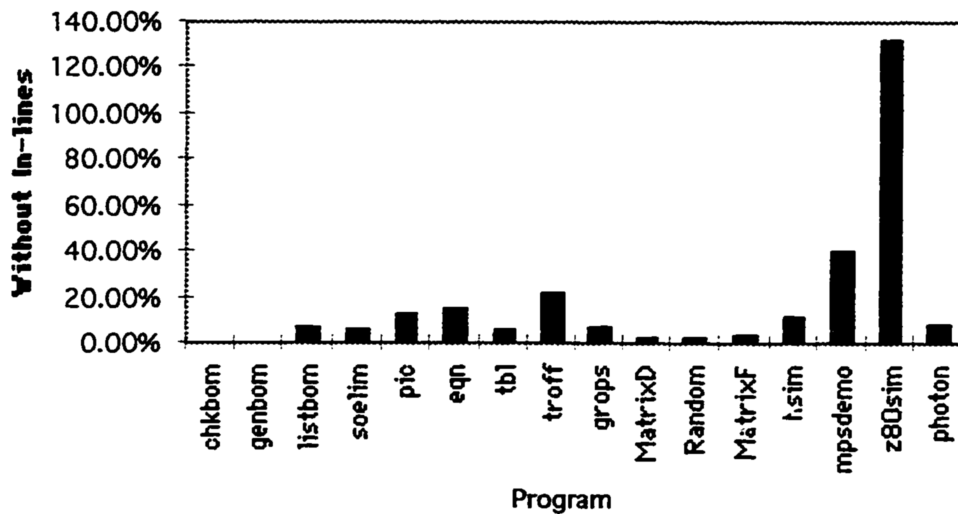


Figure 69 In-line Expansion Suppressed

different from the other programs with respect to its use of in-line expansion.

Before *z80sim* is inspected closer consider the estimated information flow added for in-line expansion turned off. Figures 70 and 71 show these estimates. Once again

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

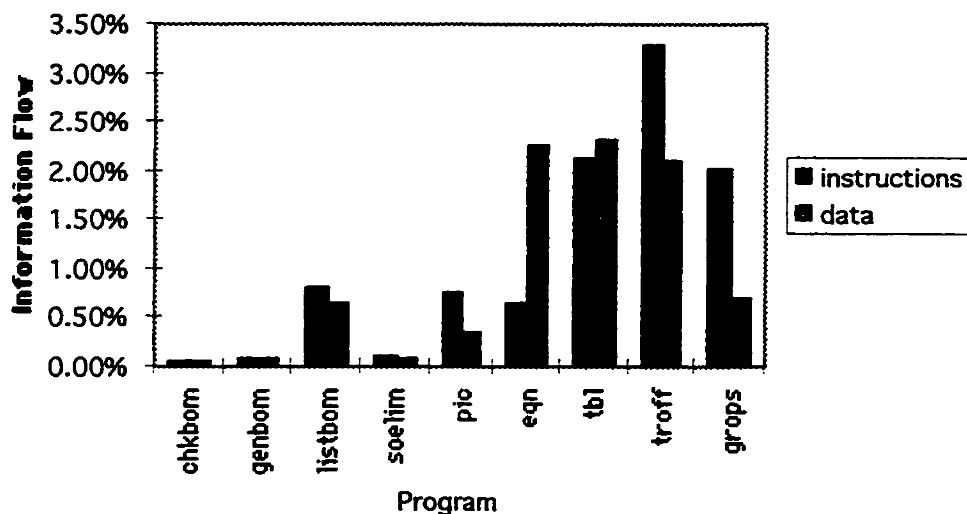


Figure 70 In-line Domain Added Information Flow

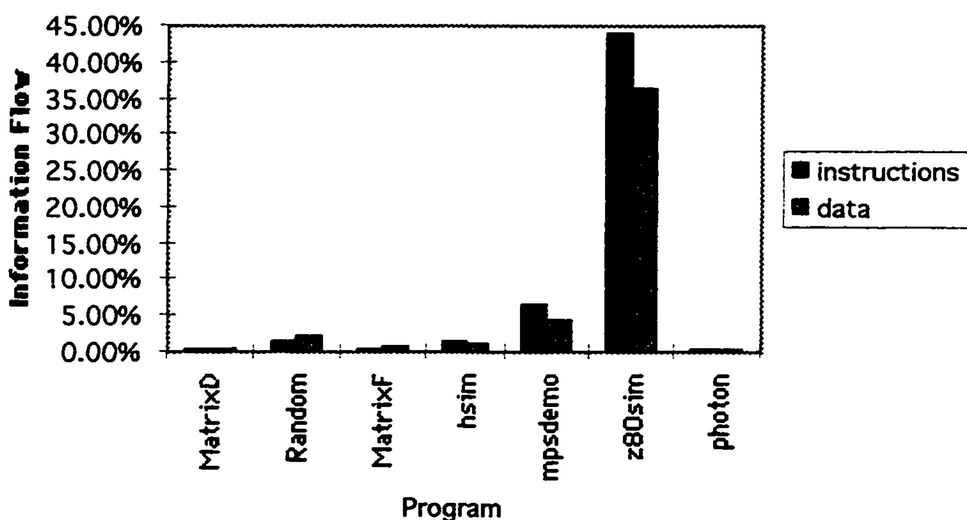


Figure 71 In-line Domain Added Information Flow

it is obvious that something is remarkably different in the *z80sim* program.

On closer examination of the source code the problem becomes obvious. The programmer defined most of the code in the header files and were intentionally designed to ensure the compiler would expand them in-line. To make this more clear

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

consider the class method definition from *z80sim* shown in Figure 72. This function is defined in the header file which defines the class *z80_cpu*. The *inline* keyword tells the compiler the programmer intends this function to be expanded in-line. Both *ldi_mc* and *intr_check* are also methods of class *z80_cpu* and are likewise declared with the *inline* keyword. The result of the program design is that all three functions are implemented in-line by the compiler. When in-line expansion is turned off the result is three additional subroutine calls. One of these will be an internal node requiring a domain save and restore. The other two, in this case, are leaf nodes. However, these leaf nodes are within the same object. Intra-object calls do not require a domain change. Hence there would be no need for the new internal node to save the domain. In this one case a single call to *ldi* in an object of class type *z80_cpu* would result in one internal node and two leaf node penalties being recorded when in fact they are unnecessary. This shows that our estimation of in-line domain changes is pessimistic.

Beyond the unnecessary domain change penalties, there is another important point to make about in-line expansion. Often times domain changes are not required at all and the compiler should recognize these cases easily. Another example, repeated often in the source code, will be taken from the *z80sim* program. Consider the code in Figure 73. The class method definition is declared *inline* to the compiler just as seen above. In this case the method takes a single argument, increments it by one, and returns the result. There is no object data involved. There is no need for the compiler to implement a domain change. The experimental data collected has no means to detect these types of routines. They are assigned penalty estimates the same as any other. This

```
inline void
z80_cpu::ldi()
{
   ldi_mc();
   intr_check();
}
```

Figure 72 In-line Example Code

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

```
inline z80_16bit
z80_alu16::inc(z80_16bit v)
{
   return(++v);
}
```

Figure 73 In-line Method Without Object Data

make the estimates very high compared to what is necessary.

Much of the performance penalty for domain changes due to in-line expansion can be argued away as not really necessary. A good argument could also be made that programmers should not design or code software to count on compiler behavior to language special features. The *inline* keyword is only a hint to the compiler, the compiler could implement the function in-line or not in-line as it pleases. In terms of the von Neumann bottleneck being exposed to the programmer, the *inline* keyword could be seen as a very bad idea. On the other hand, these arguments miss an important point. It is hard to predict what the programmer will do. For any cache design there is an algorithm that will make it perform at its worse. This could be why the capability-based machines were never successful. In the average case their performance was good, but many small protection domains could cause them to perform badly. LOTA still performs reasonably well, but the potential problem of in-line domain changes was not foreseen.

5.3.6 Summary

Now that the individual penalties have been estimated, the remaining task is to add them up and see if the overall performance is acceptable. Figures 74 and 75 display this composite. The figures present the data a little different than previous figures. The 100 percent mark is the total time for the LOTA version of the program to complete. The solid black portion of the bars represents the percent of that time which the original program required to complete. All programs have expected performance penalties of less than 40 percent. Most are in the 10 to 15 percent range. Several have

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

five percent or less.

LOTA represents a new design trade-off between hardware, software, and error containment. Its performance has been demonstrated to have the potential to be much better than the software defect detection tools even in their absolute best case. The implementation of LOTA begins with a familiar base design and adds to it in several ways, but the complexity of these additions are far less than the ground up approach in the hardware object-based systems. The object-based systems define new and complex operations that cause implementation difficulties and result in lower performance. LOTA has been carefully designed into the cache architecture in such a way as to parallel its operations with functions already required. The result should be a processor with equal cycle times.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

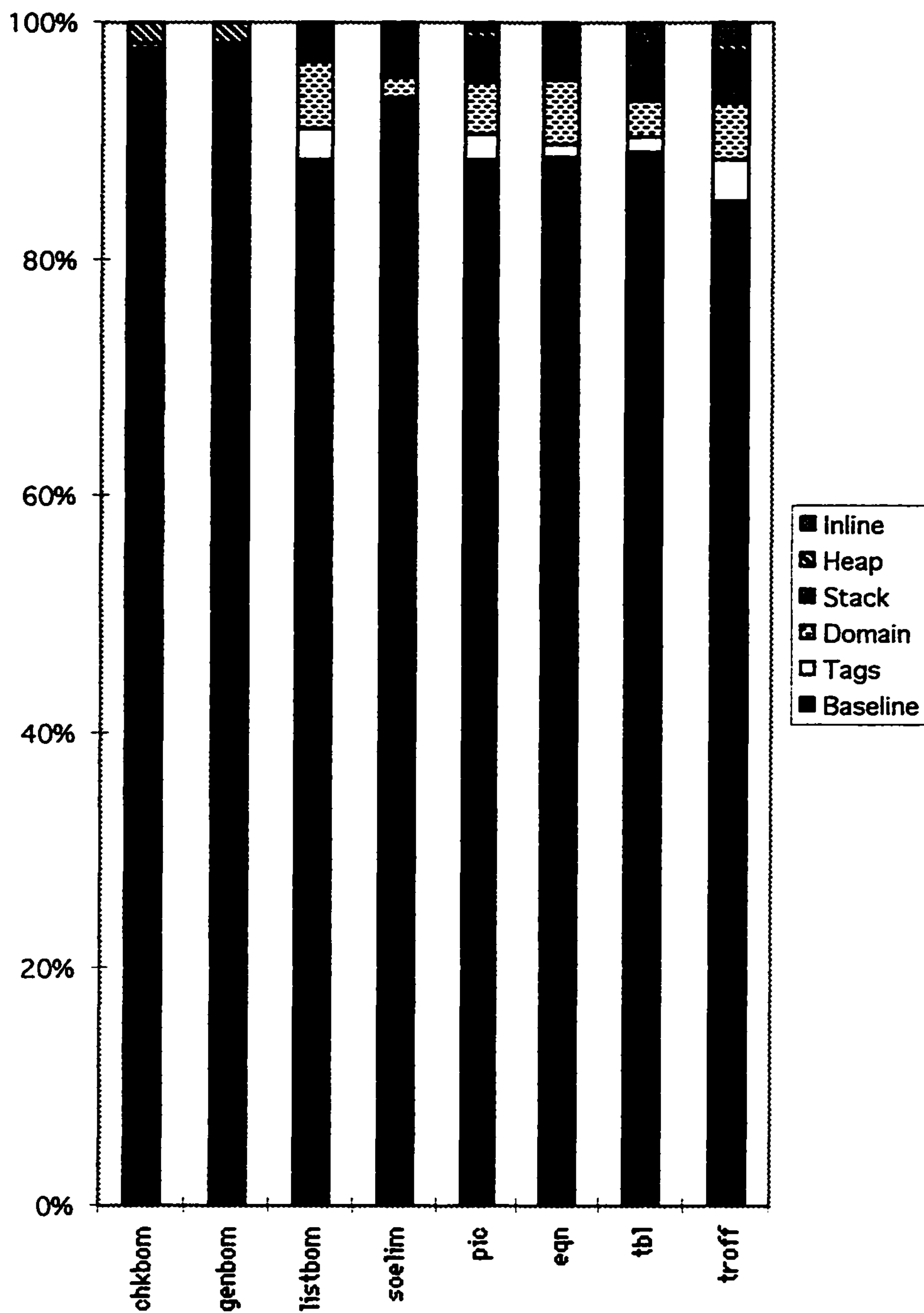


Figure 74 110L Performance Summary

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

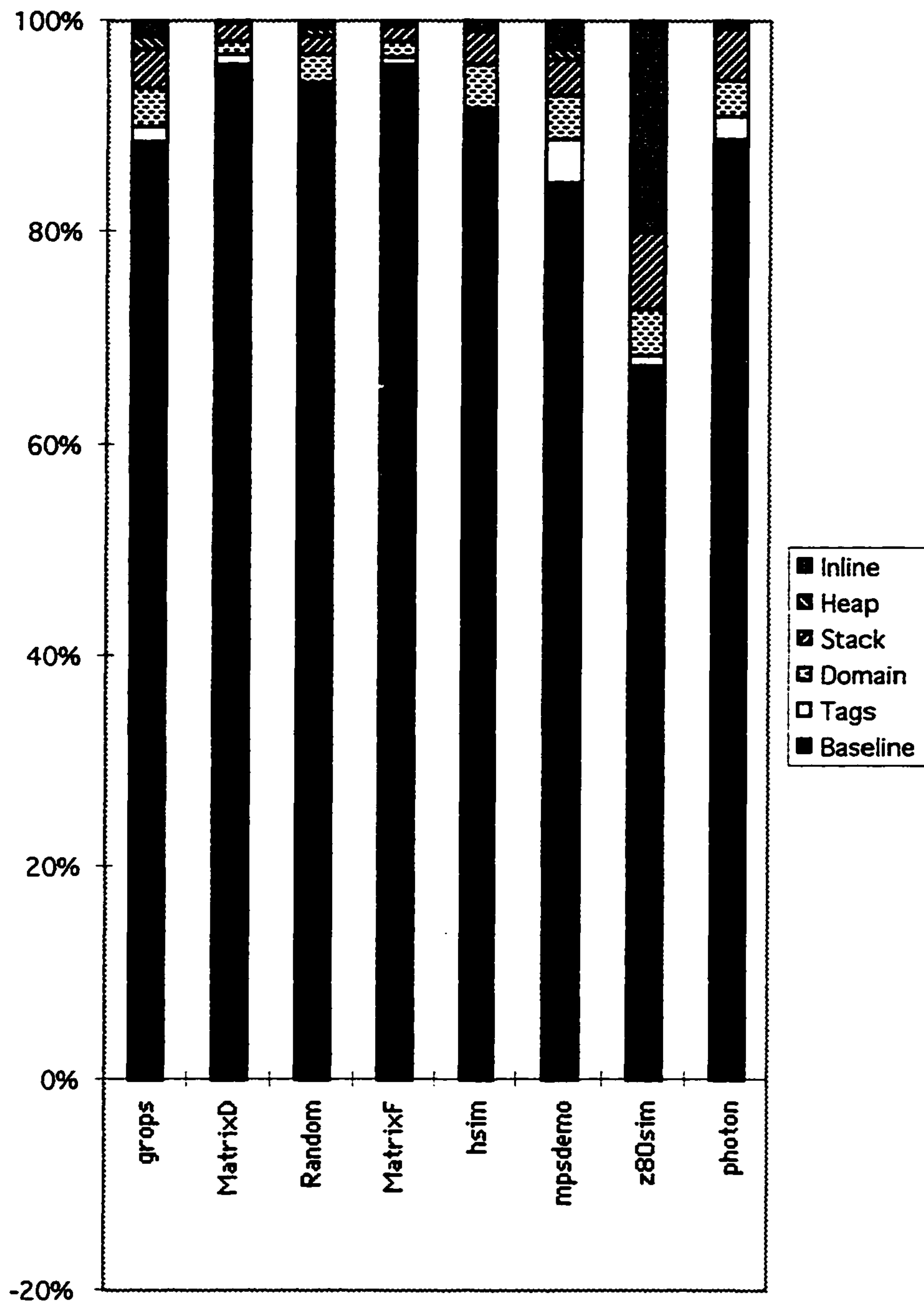


Figure 75 110L Performance Summary Continued

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 6

Refinement

6.1 Tightening the Bounds

The inherent penalties for LOTA include managing tags in heap space and stack space. The implementation penalties are the tag transfer between memory and cache as well as saving and restoring domain information across calls. The estimates presented so far have some overestimations of these penalties. This section will explore the tag transfer and domain crossing penalties to tighten the bounds on expected overhead.

6.1.1 Cache

LOTA only requires data accesses to be checked, instruction fetches do not need to be checked. In the estimates for the penalty of transferring tags the measurement did not separate instruction cache access and data cache access. The estimates will tend to be high. To get a feel for the size of the over estimate the instruction access and data access characteristics for the programs are presented in Figure 76. In all cases instruction access represents over 50 percent of all memory accesses. These numbers were generated by the XSim statistics for the number of times the instruction cache was accessed versus the number of times the data cache was accessed. The effects of tag transfer will vary with the instruction to data access ratio as well as the cache hit ratio.

Unfortunately, XSim does not distinguish instruction memory from data memory automatically. XSim does allocate simulation memory for the program under test based on the sections in the executable file. Each section can have different memory characteristics. The process of changing the characteristics of a memory segment is to modify the simulator memory map based on the information created

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

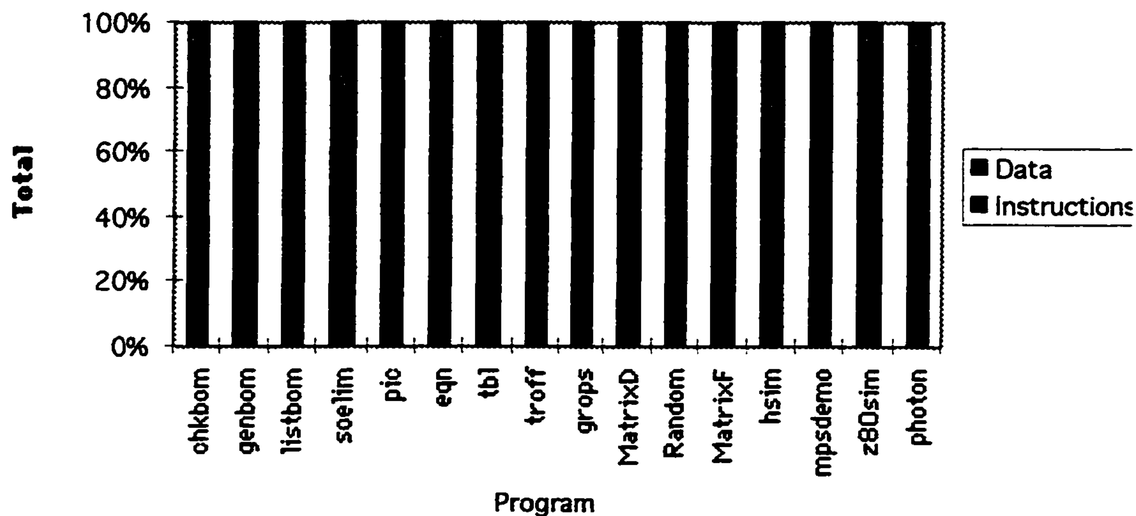


Figure 76 Instruction to Data Access Comparison

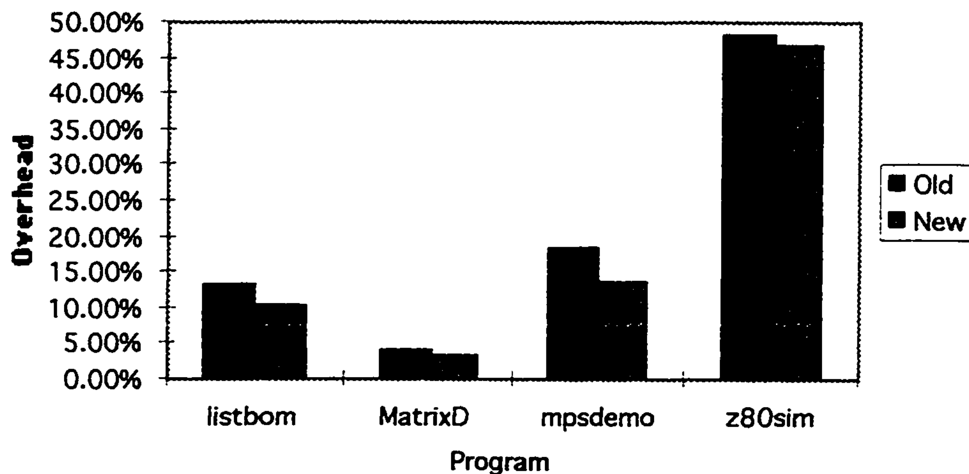


Figure 77 Performance Overhead Without Instruction Tags

when the program was loaded. The process does not lend itself to automation as easily as previous experiments. Some of the programs were simulated manually to extract their performance characteristics and are shown in Figure 77. The performance estimates do improve, but not dramatically in most cases. In particular it has not had

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

the effect on *z80sim* as could be hoped.

6.1.2 Domains

The other area where the estimates of overhead could be overstated is in the domain crossings. This is especially true for in-line domain crossing. To get a better understanding of how the estimates could be overstated it will be necessary to understand the programming language better. Consider the C++ class definition shown in Figure 78. The class is *Atype*. Once the type has been declared new *Atype* objects

```
class Atype {
        fun1();
    public:
        fun2();
}
```

Figure 78 C++ Class Definition

can be created. *Atype* is a very simple class definition. It only contains two methods (class subroutines), *fun1* and *fun2*. Notice the keyword *public* in the class definition. Only functions which are declared after the *public* keyword are visible to objects of other types. *fun2* is a public method for objects of type *Atype*. *fun1* is called a *private* method of class *Atype*. Private methods cannot be called directly by other objects. These methods are only called by other methods of *Atype*. The implication here is that all calls to private methods must be intra-object calls and would not require a domain change in LOTA.

Since *z80sim* represents the worse case for estimates of overhead it will be analyzed for the cause. XSim records profiling information. The profiling information assigns clock cycles to symbols of the program as it is being simulated. The symbols are extracted from the symbol table kept in the program file. Many of the symbols represent subroutine names. XSim also keeps track of how many times the instruction exactly at the symbol address is executed. In the case of a subroutine this number

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

represents the number of times it was called. It is possible to examine the profile information and determine the number of times a private method was called. Upon examination of the *z80sim* source code it was discovered that class *z80_cpu* had many private methods. From the profiling information from experimental run three, it was determined that during simulation these private methods were called a total of 4,854,251 times. All of these could be excluded from the domain crossing numbers. The class *z80_flags* resulted in 417,816 calls to private methods. In another case, the class *z80_memory* was derived from class *MEM_Ref*. A public definition of the array operator [] called a base class method. These calls are also intra-object and do not require domain changes. These calls accounted for 4,043,340 of the total. In one more example, a class defined a method which took no arguments and had only one statement, *return 1*. The in-line expansion of this statement is simply to throw it away. This method was called 1,155,195 times. Just in these four instances it has already been determined that 10,470,602 calls were counted as domain changes needlessly.

When a function which was in-line expanded is changed to a subroutine call, the routine which calls it must now be an internal node on the call graph. Recall that only internal nodes need to pay the domain save and restore penalty. Unfortunately it is impossible to determine which new calls are responsible for new internal nodes being encountered. Consider the case where no needless internal nodes are created. In this case all the unnecessary domain changes discovered above would cause no needless domain saves and restores. Even so, the overhead for the *z80sim* program falls from 48 percent to 40 percent. However, it is likely that some of these domain changes resulted in internal nodes paying domain save and restore penalties in the estimates. If this is true then the improved performance of *z80sim* would be even greater. If all of the domain changes were assumed to result in needless domain saves and restores the overhead would drop to 28 percent. So far only a few of the possible private method and other intra-object calls have been accounted for. There are many more. In the absolute best case there would be no need to cross domains in-line and the overhead

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

would drop to only 18 percent.

The point is, *z80sim* is a very unusual program making use of a language feature for in-line function definition in an attempt to write an extremely fast Z80 simulator. LOTA appears able to handle this worse case scenario with less than 40 percent overhead. As with all object-based systems any penalty for crossing domains will become a larger percentage of total execution time as objects become small.

6.2 Improving Performance

Separating the penalties in LOTA once again into inherent and implementation types, this section will explore one of each and offer some hope for improved performance.

6.2.1 Tag Transfer

Processor design is advancing rapidly. To improve performance of new processor designs the bandwidth between processor and memory is increasing. The two ways to increase bandwidth is to increase external bus access speeds and to transfer more information in a single bus transfer. Processors are approaching the point where they have enough pins to transfer entire cache lines in a single memory transfer. If the external data bus can be enlarged so that tags can be transferred along with the critical word then the tag transfer penalty would virtually disappear. It may not completely disappear until the bus can be enlarged enough to transfer all the data and the tag in a single operation.

6.2.2 Stack Space

One area where LOTA has an inherent penalty is its need to change the tag values as memory is assigned to different protection domains. The added information flow required for stack manipulation in LOTA is large. An idea from segmented systems might help in stack management. The observation is made that permanent data is ordinarily non-sequential. Blocks of memory are allocated as they are needed and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

come from the heap space in no obvious order. On the other hand, temporary data is sequential and there is nothing random about its assignment from stack. Another observation is that stack space is allocated in blocks and the only protection necessary is to protect earlier blocks from the current one. The current block represents the currently executing subroutine while all the blocks below it represent stack frames for all the subroutines between the current one and the root node in the call graph. Figure 79 demonstrates this view. In LOTA stack frames for different subroutines are tagged

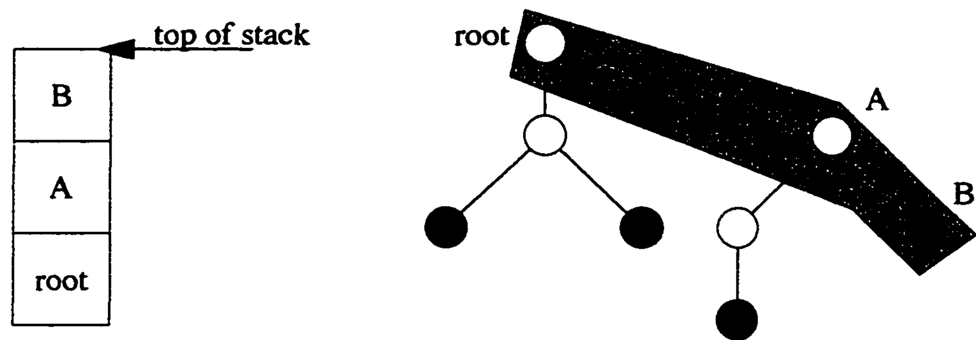


Figure 79 Stack Frame and Call Graph

with object ownership. The design allows for stack space assignment to an object to be free or nearly so. When the claimed space is returned it must be tagged again, an operation which in not free. A conceptual problem with this is its chaotic nature. The penalty is very much related to the size of a stack frame.

In the segmented approach to stack management each stack is marked by a base and limit field which the processor enforces for the currently executing object. The proposed idea to improve LOTA is to define a new field, called a stack barrier, which separates the current objects stack frame from anything below it. The stack barrier would operate like the base field of a segment register but would have no corresponding limit field. The stack barrier could therefore be defined to be another register exactly as the tag registers of LOTA. The limit field is not needed because there

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

is no stack space above the currently executing object which is claimed. This is the basic definition of a stack.

In this alternative version of LOTA, a domain change would include telling the processor of the new barrier between its own space on the stack and the space the new object might claim. Arguments to the new routine would occupy space above the barrier. Figure 80 demonstrates this new idea in the case where a method *B* calls *C*. As shown in the figure, as method *B* is operating it is limited from accessing memory for stack space below it. As it prepares to call *C* it places arguments, if necessary, on the stack. It sets the barrier between itself and the arguments for *C* during the call. Following the call, *C* can claim additional space as it needs. To manage this procedure the processor would need a new register set defining the new stack barrier on domain change. It is clear these registers could be operated identically to the tag registers already defined for LOTA. The previous barrier would then need to be saved and restored only if the routine is an internal node on the subroutine call graph. The penalty would be identical to the ones already demonstrated for tag registers.

This alternative implementation for LOTA will be referred to as LOTA+. To analyze the effects consider the stack penalty calculated earlier in terms of lines of

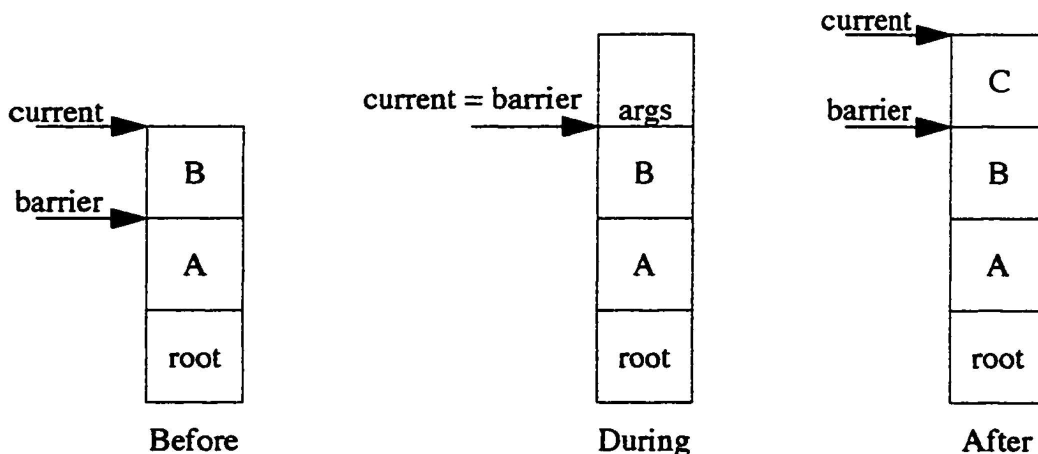


Figure 80 Stack Management During Domain Change

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

stack space allocated. The new values are calculated in terms of internal and leaf nodes exactly as they were calculated for domain changes. The comparison in terms of estimated performance penalty is shown in Figure 81. Notice how the performance penalty actually went up in most cases. This demonstrates that the original design of LOTA performs well when stack space allocated per call is small. Remember however, that LOTA pays a penalty based on lines of space allocated, a value which differs for each subroutine. LOTA+ pays a fixed penalty per call depending on if the routine is a leaf or internal node in the call graph. This value is variable based not on which subroutine is called, but how many subroutine calls are made. LOTA would tend to be better if stack frames are small, LOTA + would tend to be better as stack frames were large. Both improve with fewer calls, but LOTA+ distinguishes the intra-domain calls and pays less of a penalty. In our estimates the lines of stack is accurate, but the internal and leaf nodes overstate the domain crossing requirements.

It is interesting to look at LOTA and LOTA+ in terms of information flow as well. Figure 82 shows the comparison in terms of instructions while Figure 83 shows the comparison in terms of data. Notice the instruction penalty for LOTA is usually

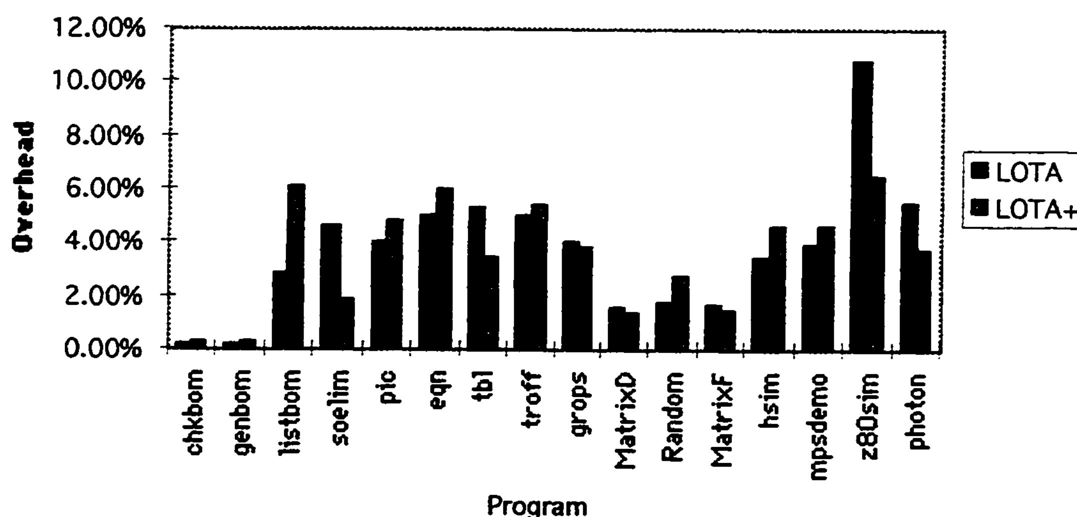


Figure 81 Stack Penalty Comparison for Domain Crossing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

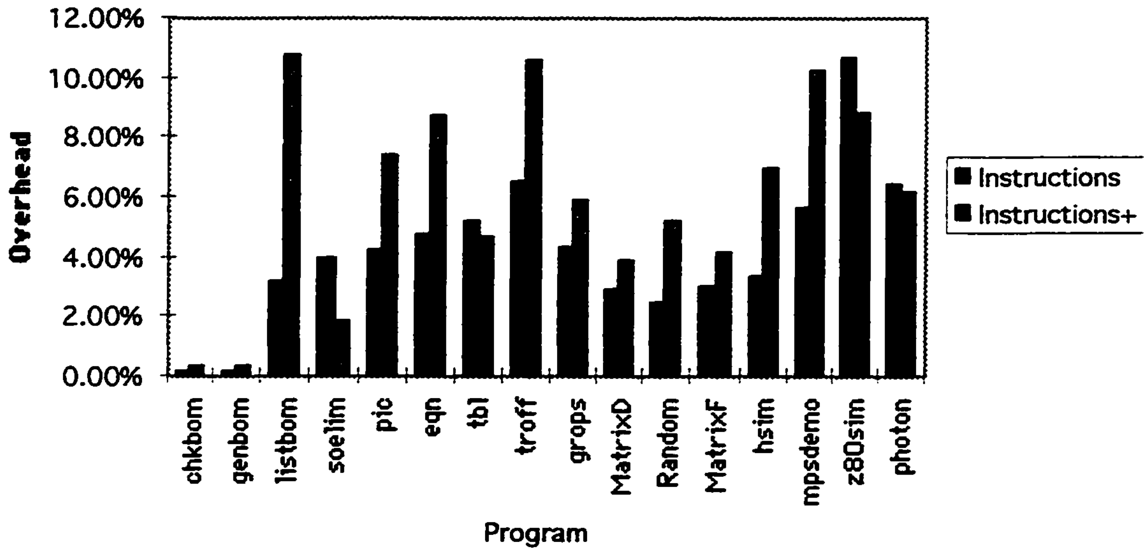


Figure 82 Information Flow Comparison for Instructions

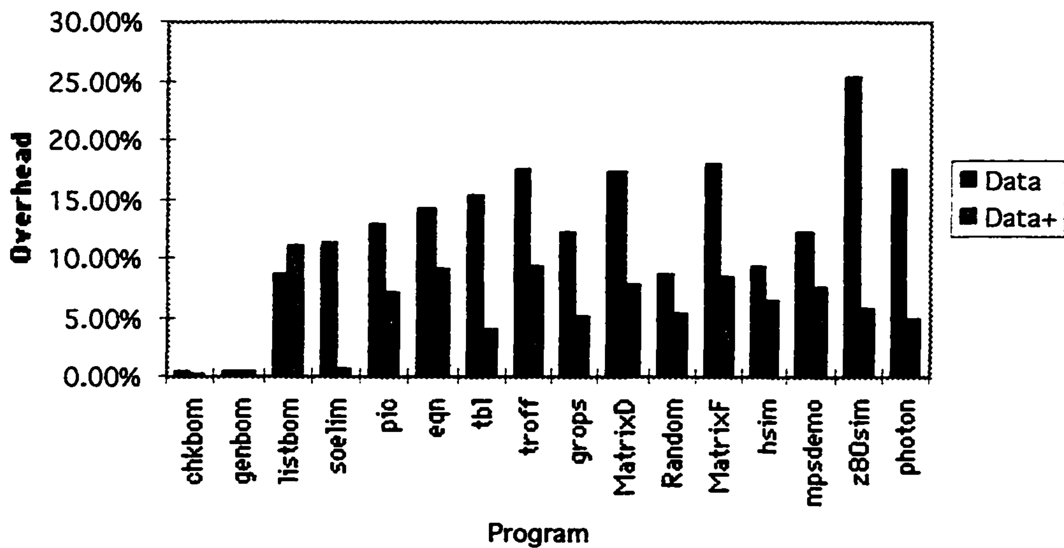


Figure 83 Information Flow Comparison for Data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

smaller while the data penalty is often larger. This demonstrates the trade-off more clearly for small stack frames, LOTA+ uses more instructions with less data traffic. In LOTA all the instructions require a corresponding data transfer.

6.3 Improving Security Against Defects

The LOTA protection domains are compiler controlled. The value used to define a domain for the currently running object is stored in a special register. As such it is out of scope for corruption by ordinary software defects. There is no code in C++ a programmer could use to directly effect the value in the tag registers. In many cases the compiler can generate the address directly. In the 88110 a common sequence for the compiler to generate is:

```
or.u  r3,r0,<immediate 16>
or    r3,r3,<immediate 16>
```

This code uses 16 bit immediate values in the instruction set to generate half of the target address with each instruction. The *or.u* instruction generates the upper 16 bits of the target address while the ordinary *or* instruction adds the lower 16 bits. This type of target address calculation is safe from program errors since it is contained entirely in the instruction memory. However, there are cases when the compiler cannot know at compile time what the target address will be. In these circumstances a data memory location is used as a pointer to the object. Since these values are used to define domains, then a program defect could cause an incorrect domain value to be loaded into the tag registers. Notice that it will not effect the current object, but the called object. Also, it is improbable that the value would be a valid domain. If it was a valid domain it would be even more improvable that the called object would generate an incorrect address and access the wrong memory. Most likely is that a protection fault will occur in the target object on its first memory access.

No matter how improbable, it still might be advantageous to close the security hole. There are two requirements which must be met to have better security. The first is that object entry points and domain values cannot be altered by a program without

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

detection. The second is that the return address and domain value for subroutine calls also be protected from undetected modification by the program. Together these mechanisms bind target addresses for domain calls and returns with the protection domain values.

6.3.1 Domain Entry Points

The entry point and domain value for an object could be kept in an object function table. A function table is a table with pointers to the functions (subroutines or methods) for an object. The processor would need instructions which allow these target pointers to be loaded along with the object tag in an operation which is inseparable. The 88110 already allows for double words (64 bit words) to be loaded in support of floating point values. It would be possible to define a new load instruction which transferred two 32 bit values representing the target address and protection domain in a single 64 bit load. The hardware would have to maintain the integrity of this pair of registers to detect any tampering by the program. The register pair could then be used to define a new *enter protection domain* instruction.

The jump table would have to be manufactured at the time an object is created. This would require a privileged memory manager capable of generating domain values and initializing tag values outside the scope of normal LOTA definitions. This would require a facility to identify such a memory manager, but it should not be an obstacle. Of greater concern is the additional overhead involved with manufacturing jump tables and the indirection involved with domain calls. These would be significant, but perhaps less costly than the mechanisms in previous object-based systems.

6.3.2 Domain Return

Along with the binding of entry points to domains, there must be mechanisms to allow saving and restoring domain values. The domain save and restore must have the additional capability to bind the return address with the domain value. In the 110L the return address is copied into general register r1 while the return domain value is copied

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

into t2. The integrity of these two values would need to be maintained by the hardware. These values will need to be saved into the stack to make additional calls. A double word store could be used to store them as a pair. However, the location these values are stored in must maintain their integrity. This would probably require a separate cache line tagged with a value to denote a return address and domain pair occupy the space. A subsequent load operation to restore the value could make integrity checks to assure the values have not been tampered with. The return from protection domain instruction would then be assured as a minimum that the target return address is a valid one defined in a previous call and that the corresponding domain is correct.

This discussion is only intended to provide the outline of how LOTA could be expanded to incorporate a more secure protection mechanism for object encapsulation. This does not necessarily mean such an implementation would be space or time efficient, only that it could be done and it might be worth the effort.

6.4 System Issues

Up to now only process and memory issues have been addressed by the design and analysis of LOTA. This section will provide a brief overview of the major system issue, namely disk transfer of data which includes tags. Then a description of alternative implementations of tags is considered.

6.4.1 Disk Transfer

One of the important concerns in LOTA will be the transfer of memory tags to and from secondary storage. For this discussion the 110L will be used. The 110L has cache lines which are 32 bytes in length and are tagged with a 4 byte tag. Most computer systems utilize a direct memory access (DMA) controller for transferring data between memory and secondary storage through a disk controller. The DMA and controller are often integrated. The DMA function must be aware of tagged memory and have the capability to transfer tags and data. It is operating system software which provides the controller with instructions to transfer memory. These instructions could

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

easily be augmented with tag information. The controller would need to be altered to have the capability to transfer the tags optionally.

DMA operations occur in multiple of bytes. Some transfer single bytes, some double bytes, and others full 32 bit words in four byte quantities. Tags are multiples of any of these size quantities so there is no problem with DMA operations. Disk sectors are most commonly 256 or 512 byte quantities. This represents a compromise between efficient transfer and optimized use of space. The operating system also divides memory up into pieces called pages. For the 88110 these pages are defined by hardware to be 4096 bytes long and the operating system must use multiples of this size for its own pages. Therefore it takes eight or 16 disk blocks to save one 88110 page of memory. In LOTA these 4096 byte pages, for data pages, must have 128 additional bytes of tag memory. For a balance to occur the operating system could use 8192 byte pages and 256 byte sectors. This would require 33 disk sectors for every operating system page when tags are required and 32 disk sectors for pages not requiring tags. Since all of the quantities are power of two values there should be no alignment constraints with such an implementation.

6.4.2 Tag Memory

Much of the tag space remains unused. During the operation of a computer with LOTA the tags would be used sometime and not used at other times. Only the data pages of a program utilizing the object-based protection would have their corresponding tags utilized. This gives rise to the possibility of reducing the number of tag bits in the system. Three methods to this end will be briefly discussed.

6.4.2.1 Partially Filled Tag Space. To reduce the amount of memory required for tags in the memory system it is possible to simply populate only part of the tag memory space. Only pages which have their associated tag space occupied by real memory would be candidate memory for object-based data pages. This would be practical in a system which is only partially occupied with object-based programs. The operating

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

system would need to be aware of which pages were tagged and give bias to their use in object-based programs. There appear to be no problems in hardware for partially populating the tag space but the operating system complexity would be increased to manage available memory.

6.4.2.2 Separately Mapped Tag Memory. Another possibility would be the separate mapping of memory. One mapping of memory would appear to programs and the compiler exactly as described for LOTA. Another mapping of memory would allow the operating system to access tag memory just as it would any other memory. The address range which allows the direct access to tag memory could be mapped out-of-range to ordinary programs. To the operating systems and the hardware controllers this second mapping would significantly reduce the complications of handling pages with tags. The down side to such an approach is that memory decoding is more complex and the address space available is correspondingly reduced.

6.4.2.3 Reconfigurable Tag Memory. A third possibility would be to use a second memory mapping unit similar to the memory management unit. This second memory mapping unit would map program addresses to the tag space while the normal MMU mapped program addresses to data space. Such a hardware mechanism has been proposed in [64] and is called the Reconfigurable Tagging Architecture (RTA).

The RTA allows any memory in the system to be optionally used as tags. When tags are needed the memory is allocated for the tags at the same time that memory is allocated for data. The RTA requires the processor to have a separate translation unit and cache for tags. This would be a significant departure from what has been presented for LOTA up to this point. The advantages of such a system would be maximum utilization of memory without the complexity of tag space separate from storage space. Most of the complications of managing processes and memory with tags disappear. Another benefit to LOTA is that the design decisions for tag support for a process is separated from the data cache design decisions. The amount of data which is tagged

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

could even be put off completely to allow the run time system to pick an optimal value.

Of course there is some complexities involved with the RTA. The processor must be designed with an entire new unit. This unit must contain cache memory for the address translation as well as for caching the tags. A whole new set of parameters will determine the performance of such a system. The RTA does offer the greatest flexibility in the design of LOTA.

6.5 Application of Tags

The most common software defect which was recorded in Chapter 2 was the uninitialized memory read error. Spare tag bits in LOTA could be used to help trap this defect type. Recall that domain values in LOTA are ordinary addresses. These addresses are 32 bits in length. However, a cache line is 32 bytes in length and therefore five address bits are unnecessary in LOTA. Tags in the 110L therefore only need be 28 bits since one bit has already been defined for use as a global flag. The remaining four bits could be used for other purposes. Two possibilities are presented.

6.5.1 Uninitialized Read Tags

With the four bits of available tag space LOTA could be enhanced to detect uninitialized memory read operations on 64 bit boundaries. This compares with the 8 bit boundaries that Purify can detect, but otherwise is the only system presented which can detect such errors at all. The four bits would be assigned one each to a pair of data words in the cache. When allocating memory from heap these bits would be cleared when the tags are initialized. A memory write operation to any bit in the 64 bit quantity would set the associated initialized tag bit. Any read from a word which did not have its initialized tag bit set would be flagged an uninitialized memory read error.

6.5.2 Boundary Condition

A competing idea for the use of the four spare tag bits would be in setting a boundary condition in the cache line. LOTA requires memory to be allocated in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

multiples of cache lines. There will be times when the space allocated is larger than necessary. Access to this additional space is not an error which is ordinarily caught by LOTA but which may represent a software defect. The four bits of available tag could be used in LOTA to set a boundary condition in the associated cache line. With four bits the granularity could be set to 16 bit boundaries. Any access beyond the boundary would be flagged as an invalid memory access much in the same way that Purify detects an array bounds access error.

6.6 Alternative Applications for LOTA

LOTA was designed to be an extension to a processor architecture. Although the changes required to the base architecture are meant to be feasible, it is still a significant change. This section will present two possible alternatives for an implementation of LOTA which may provide more attractive cost versus benefit characteristics.

6.6.1 Second Level Cache

As presented, LOTA limits the memory which an object may access. The stated goal is to bound defects to object boundaries. To reach this goal the only necessary feature is to limit the memory write operations an object makes. A memory read operation outside the scope of an object's domain does not cause a defect to be propagated to another object directly. The only defect it can propagate due to a memory read error is an incorrect result. A memory write error on the other hand can directly change another object causing it to be defective as well. This observation can lead to a relaxation for the boundary checks for object access, where only write operations are limited to object boundaries.

When memory write operations are limited to the memory an object owns, many of the hard to trace software defects are still trapped. Any array bounds write error, free memory write error, dangling pointer error, or corrupt pointer error which result in a write to memory outside the domain will be trapped. No read error will be trapped unless the read operation is invalid to the entire process. A read of instruction memory

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

is such an example. This relaxation of the requirements leads to other possible implementations of LOTA which may be more practical to implement.

The cache in a processor must have good performance. It must have a sufficient hit rate and cycle time to meet performance goals. LOTA was designed to avoid impacting the cycle time of the cache but it necessarily increases the size of the cache. The important characteristic of the cache in LOTA is that domain access rights are checked. These checks occur on read and write operations. If LOTA was changed to only require writes be checked then the place in which the check occurs could be pushed off the processor chip into a second level cache. The primary cache is still assumed to be on the same chip as the processor. All read operations are free to be performed in this cache. To provide write checks the on chip cache must be managed with a write-through policy. The write-through policy requires that all write operations must cause an external bus write to keep the external memory system up-to-date at all times.

The 88110 processor has a companion part, the 88410, which provides control for a second level cache. A block diagram for a system with the 88410 is shown in Figure 84. The *62110 array* in the figure is a memory array made up of dual bus 62110 chips to implement the data storage for the cache. The 88110, 88410, and 62110 were all designed as companion parts for system construction. The 88410 can control a second level cache from one quarter to a full megabyte in size. The 88110 and 88410 were designed in such a way that the second level cache always contains a strict super set of data in the 88110's cache.

The 88410 contains the address tags for cache lines while the 62110 array contains data only. In a LOTA system the 88410 would be expanded to include the domain tags as well. The tag register of LOTA could be implemented as normal memory mapped control registers in the 88410. To update the tag registers the compiler could use *xmem* instructions. *xmem* exchanges memory with a register value. The importance of *xmem* in a LOTA implementation is that these instructions are not

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

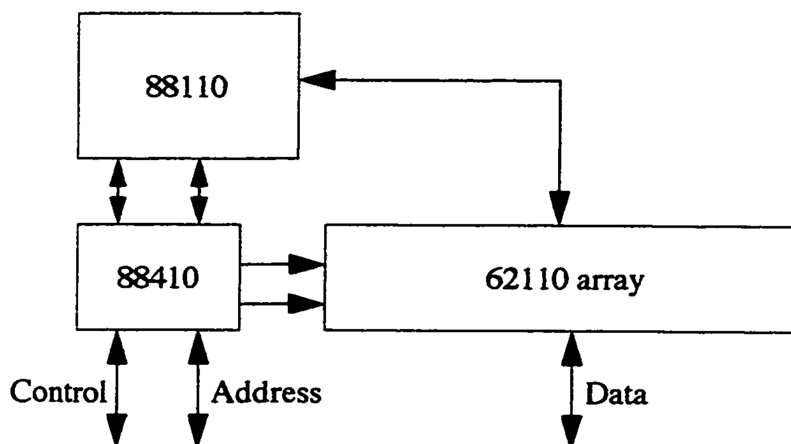


Figure 84 88110 System with 88410 Second Level Cache

ordinarily generated by compiled code and the instruction has the side effect of serializing the 88110. Serializing the 88110 means that all previous instructions must complete before the next instruction executes. Although this would be a performance penalty, it would allow the 88410 to remain synchronized with respect to the domain of the currently executing object.

In this implementation of LOTA the compiler would generate an *xmem* instruction prior to making a subroutine call which needs a domain change. The serialization effect would cause all previously issued instructions in the old domain to complete before the new domain is installed. The subroutine called would then have all its memory accesses checked in the new domain. The write-through policy of the primary 88110 cache would assure the 88410 had the opportunity to flag any write access with a domain violation. A read access which missed in the 88110 cache could optionally be checked for domain access rights as the 88410 supplies the data.

An important characteristic of secondary caches is that they are optional to a system. Many popular computer systems have second level caches implemented as a plug-in card that can be installed or removed after the system is shipped. It could be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

possible to design such a card that has the entire implementation of this modified LOTA. The card would contain the modified 88410 second level cache controller and 62110 data array, but would also contain the memory for tags in the main memory system. Figure 85 provides a block diagram of such a possible second level cache card.

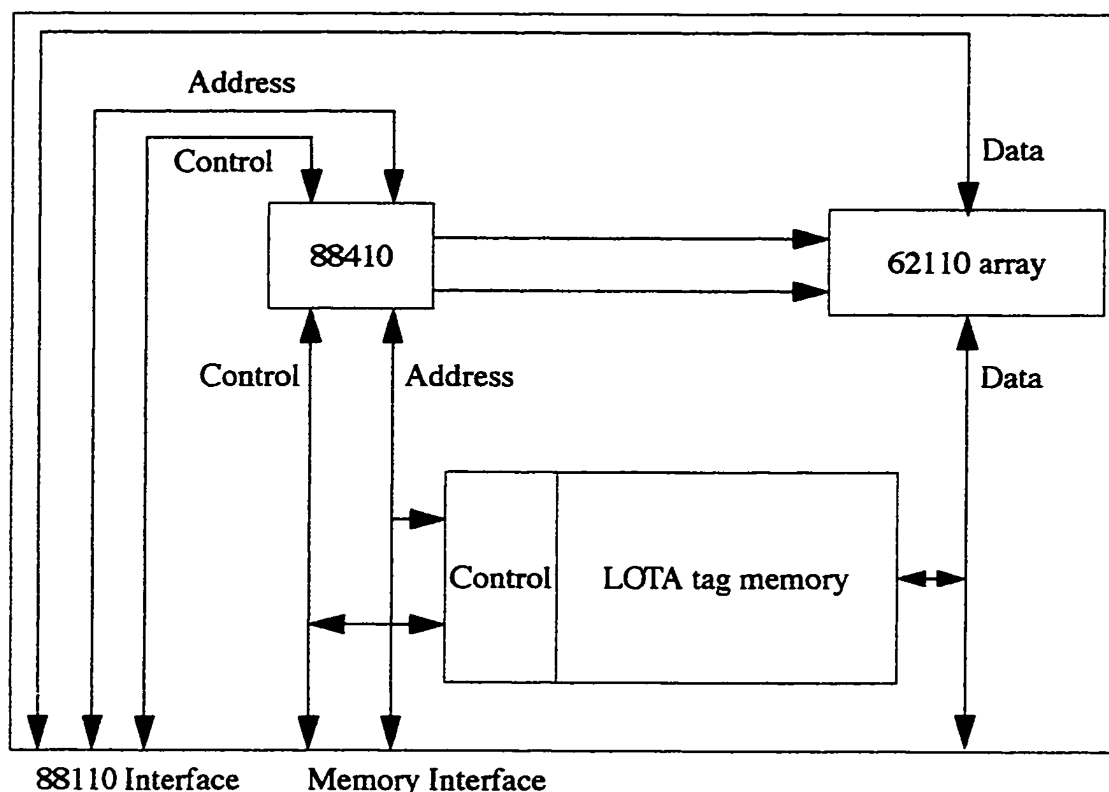


Figure 85 LOTA Implemented in Second Level Cache

This implementation of LOTA would make the entire tag support subsystem an optional piece of hardware which could be installed after a system has been shipped. The cache card would need to be designed so that operating system software can access the LOTA tag memory as ordinary memory for the purpose of saving and restoring tags to disk. The benefits versus expense in such a system may make this implementation of LOTA more attractive.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

6.6.2 Smart Card Memory Protection

In some emerging new fields there is no history as to which type of memory management hardware will prove best. The smart card field is such an area. A smart card in this discussion is a credit card with an integrated circuit embedded in the card plastic. The smart card is just recently evolving to where entire computing facilities on a single chip is integrated into the card. The next step will be to provide a small operating system kernel which allows multiple applications to securely share the card resources. The question still remains as to what type of memory management hardware will provide the security facilities required in the smart card environment with the minimum of implementation resource requirements.

The smart card operating environment is much different than in general purpose computers. The card is active only when inserted into a card terminal. Commands are received from the terminal and the card provides a response. The implication to this model is that only a single software object need to be active at one time on the card and it operates to completion and terminates. In effect a smart card application consists of objects on the terminal and objects on the card.

A block diagram of a possible card implementation is provided in Figure 86. The smart card has three connections to the terminal: data, reset, and clock [65]. There are three different types of memory in a smart card. Random access memory (RAM) provides temporary storage. Read only memory (ROM) includes all data and instructions programmed at the factory. Electrically erasable and programmable read only memory (EEPROM) is a type of memory which retains its contents without power yet can be changed by software. Although EEPROM is writable this operation requires more time and power to accomplish. For this reason data in EEPROM are ordinarily copied to RAM to be operated on and then returned to EEPROM for permanent storage. This is only one type of smart card computer organization but will suffice for this discussion.

The memory management for the smart card could be implemented as a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

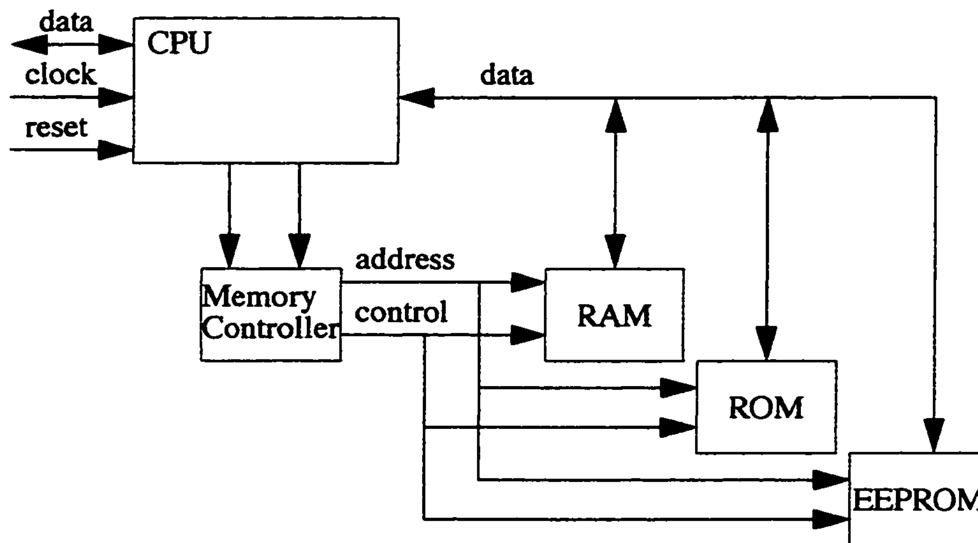


Figure 86 Smart Card Block Diagram

traditional MMU in the CPU or it could be implemented as a special circuit in the memory controller. The CPU implements supervisor and user privilege states similar to general purpose processors. The memory management policy needs to assure user privileged software only accesses memory it has rights to. Supervisor privileged software has access rights to all memory. Notice the block diagram for the smart card computer included no cache. It is unlikely that cache will fit into the transistor budget constraints for smart card applications.

The data flow through the smart card is simple. A message is received by the smart card operating system. The message specifies an object, a method within the object, and the arguments to the method. The operating system is responsible for making sure the protection domain is enforced for this object and passing the arguments to the correct routine and returning the result. This is an over simplification of the operation of a smart card. The purpose is to convey that a single object is active at one time and it completes its operation and returns a result before another object can be made active. This is a very different model of sharing than a time shared computer system in the traditional sense.

LOTA could be applied as the general purpose security mechanism for this smart

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

card system model. Consider an example where memory is organized as 32 bit words and each word has an additional four bit tag. All memory would be tagged for security reasons in a smart card. Four bit tags per 32 bit word represents a 12.5 percent overhead in storage requirements. Four bits would allow 16 applications to share the smart card at any time. The limited resources and strict security environment in a smart card application make more than 16 applications impractical. The tag bits required could be reduced by increasing the number of words tagged by a single tag value. Four words per tag would represent just 3.125 percent extra storage.

In the LOTA smart card all user privileged data access would be checked for protection domain access rights. The operating system would be responsible for assigning memory to a protection domain by initializing its tags. The overhead involved with tag initialization by supervisor privileged code would be acceptable because this operation in a smart card is very rare. Typically memory is allocated to an application (object) at the time the application itself is loaded into the card. The only exception is memory in RAM. But since RAM is temporary memory and only one object is active at a time, RAM can be assigned to the one active object and it need not be shared or protected. The possible exception is that the operating system may require some RAM protected from user privileged software.

The description of LOTA for smart cards has been brief. The purpose is to provide motivation for the practice application of LOTA and provide arguments for its feasibility. In the smart card environment an application and an object are one in the same. Domains are managed by the operating system and user privileged software are restricted to memory access in its own protection domain. The limited sharing and dynamic memory allocation requirements in the smart card reduce the implementation requirements for LOTA. The operating system overhead for setting up a protection domain should be very small. LOTA should provide a practical protection domain mechanism for smart cards.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Chapter 7

Summary and Conclusions

A recent issue of the *Communications of the ACM* was devoted to the software debugging scandal and its introductory article made several good points about the state of the debugging process [66]. Some of the early statements were "First, computer programs often don't work as they should, making software development costly. And too much buggy software reaches end users, leading to needless expense and frustration." These two statements quickly sum up the dark side of the experience for developers and end users. Computers are getting much faster with better human interfaces, larger memory systems, and larger disks. It is time to start putting some of these extra resources to work at improving the debugging process. "Let's make the computer take an active role in helping the programmer deal with complexity." Formal techniques to program verification are often used to support the notion that debugging is virtually unnecessary. The argument in [66] disagrees with "Not only is total elimination of bugs an unrealistic goal, but software is continually evolving. Debugging tools are as necessary to the incremental evolution of software as they are for finding errors."

7.1 The Software Problem

An original survey of software defects was presented in this research [3]. This survey used the defect detection tool Purify to instrument several programs. Data on several common types of software defects were compiled. An analysis of the data was provided which demonstrates the pervasive problem of software defects reaching the end user. Data on over 25,000 access errors representing 209 source code statements were collected. Both C and C++ software was used for this survey and a comparison of errors between C and C++ was provided.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Eliminating defects before a product reaches the end user is still a necessary goal, but it is evident that bugs will continue to reach the end user. In this era where quick time to market determines the success of a product, pressure to get products out quickly will likely cause more software defects to reach end users. There is evidence of this phenomena already where minor updates to fix bugs come at an alarmingly rapid rate. In the world wide web product arena, market demands require new features faster than bug fixes can be delivered for older products. By the time a product reaches production release its successor is in advanced customer testing.

An objective observer attempting to determine the strategy used in the market to eliminate software maintenance might conclude it was to release new products fast enough to obsolete the old versions before repair becomes necessary. Still, fixing defects after product release is expensive. Tools such as Purify are helping reduce the cost of eliminating defects before a product is released. But these tools are too expensive to be used by end users. The tools for field assistance in detecting and analyzing defects are still severely lacking. Few tools are aimed at both development and maintenance in the software life cycle. Most software maintenance requires symptoms of a defect to be reported back to the developers so that the problem can be reproduced. The question is, can more symptoms be made observable?

7.2 Logical Object Tagging Architecture

The Logical Object Tagging Architecture (LOTA) was presented as a possible solution to some of the software defect problems. This new architecture is based on two novel ideas. The first is that data is tagged with a value which identifies the object which owns it. The second is that the data cache checks for object ownership on each data access. The goal of this new architecture is to efficiently support enforcing object boundaries over the complete software life cycle. Enforcing object boundaries in hardware will trap many of the important software defect types. Providing the checks in the data cache provides the necessary efficiency.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

LOTA defines features to allow the compiler to manage object-based protection. Simple to use domain registers allow the compiler to convey to hardware the protection domain for an object on each call. Rules for managing tags are straight forward and easy to implement. The use of tag registers to define protection domains places the values out of scope of ordinary software defects. The result is a reliable and easy to manage system for the enforcement of object boundaries.

An analysis of LOTA was provided by using the 88110 processor as a base design. The features of LOTA were mapped onto the 88110 in a way which fits well with the original design. All changes to the 88110 were designed to be feasible. The modified 88110 was analyzed by using XSim, the 88110 cycle accurate timing simulator. XSim was modified to collect information useful to estimate the performance of a LOTA implementation. The parameters crucial for such estimation include statistics on tag transfer, stack space allocation, heap space allocation, and subroutine calls.

An analysis of the data collected from simulation indicates LOTA provides very good performance in most cases. In-line expansion of object methods in C++ proves problematic. But even in a particularly bad case, overhead was shown not to exceed 40 percent. The average case over the programs tested was 12 percent with several cases less than five percent. These estimates are necessarily pessimistic. All subroutine calls were counted as domain changes and assigned the penalty. Intra-object calls do not need domain changes and could be eliminated from the estimates. Some subroutines do not access any private object data and also do not require a domain change. Even in the worse case LOTA could be argued as worth the expense given the benefits.

7.3 Future Directions

LOTA was designed from the outset as an enhancement to a general purpose processor. As such, its design was constrained to choices which could ordinarily fit into the architecture of existing processors. Features which would require special support

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

were avoided. Given a clean slate, LOTA could be designed in a way to lend itself to more efficient implementations. Additional instructions which operate on data and tags could make tag management more efficient. Additional support for stack operations which understands tags could also help, especially for domain crossing. LOTA has a relatively high increase of information flow with respect to data when handling the stack and making domain changes. This could be cut in half with an instruction to handle return addresses and return domains as a single value. A tagged stack register for referencing information on the stack could also greatly improve stack management efficiency.

The design constraints on LOTA were imposed so that a realistic implementation could be derived and allow accurate performance estimates from simulation. Even with these constraints performance is very good. Without these constraints performance would be better. A restricted version of LOTA was presented which could be implemented as a second level cache card. The result could be a cost effective hardware implementation which still provides important software defect detection capabilities. In new areas for computers which are self contained and very resource constrained, such as multi-application smart cards, a variation of LOTA could provide the basis for the entire security system.

7.4 Conclusion

Tagging data with object identifiers is a new way of thinking about enforcing access rights to data. The data cache is an efficient place to enforce access rights based on ownership of data. Analysis has provided supporting facts to claim the method has merit for its application to solving part of the software defect detection problem over the software life cycle. Future direction possibilities have been given for the architecture and the analysis provided by this research indicates pursuit of these possibilities hold much promise.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Appendix A

Full Text of Electronic Correspondence

A.1 Henry Baker

From - Wed Apr 23 09:11:32 1997

Newsgroups: comp.arch

From: hbaker@netcom.com (Henry Baker)

Subject: Re: Naive Java question: Array index checking

Content-Type: text/plain; charset=ISO-8859-1

Message-ID: <hbaker-1804970757590001@10.0.2.1>

In article <zalmanE8t8sD.In@netcom.com>, zalman@netcom.com (Zalman Stern) wrote:

> Yes, I was being silly. But if this is as extreme as Henry makes it out to
> be, a system without the checking offered in 16-bit segmented environments
> would be such a step backwards that noone would use it. I'm driving home
> the point that the priorities are otherwise.

I'm having a hard time equating 'essential to do array bounds checking' with 'advocating 16-bit segmented address spaces'. I think that 16-bit segmented address spaces are silly, and they only appealed to IBM because IBM was afraid that the PC would take away some of their business from below. IBM was right, and underestimated the cleverness of the PC programmers who managed to get the brain-damaged 8086 architecture to do something useful anyway. But one should not mistake this cleverness as a vote _for_ the 8086 architecture. The _only_ thing that programmers admired about the 8086 architecture was its price relative to its competitors at the time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

As has been pointed out already, array bounds checking has 2 parts -- getting the information about the array bounds themselves, and then performing the array bounds check.

The array bounds must be stored somewhere, and if the array bounds check is to be performed quickly, this means some sort of a high-speed register. So this requires holding onto a register -- often throughout a loop. So one cost of array bounds checking is some slight additional integer register pressure. Note that languages allowing arbitrary _lower_ array bounds double this cost. Since array bounds in most sane languages remain constant (at least for large periods of time), the management of these constants can be readily performed by a modern compiler.

The other cost of array bounds checking is the actual comparisons. For the simplest linear algebra expressions (A[i,j]=A[i,j]+B[i]), these comparisons can be moved out of the loop, which then allows the bounds themselves to be moved out of registers, if necessary.

However, in the case of things like hash tables, where you have expressions like foo[<expression> MOD n], you must either have an extraordinary compiler that knows that (a MOD b)<|b|, or you have to insert a check. (The fact that languages like C and Fortran still allow expressions like (a MOD b) to return negative numbers for positive b has irked me for 25 years, even though this actually _reduces_ performance due to the following array bounds check. And yes, nearly every single software person has tripped over this particular bug in his programming life -- usually during 'system integration', when the costs are highest.)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

For arrays used as other kinds of tables -- e.g., in graphics -- wherein
you do a float calculation, and then convert to an integer and then index
to an array, it is hopeless to expect a compiler to _prove_ that the
array bound can not be violated.

If the actual array bounds check can be done completely in parallel with
other calculations, then it may cost nothing. However, a serious question
arises about what sort of interrupt to cause if the bounds don't check.
One would still like to find out where in the code the program went wrong,
and which array was about to be violated. If such an exception wants to
be 'precise', I can imagine that this precision may exact additional costs,
but this depends critically on the architecture.

A.2 David Chase

From: David Chase <"mylastname "@ world dot std . com>
Subject: Re: Naive Java question: Array index checking
Date: Mon, 21 Apr 1997 02:28:49 GMT
Reply-To: mylastname, @, world, dot, std, ., com
Content-Transfer-Encoding: 7bit
Content-Type: text/plain; charset=us-ascii

Andy Glew wrote:
>
> Will somebody please write a C language to C language preprocessor
> that adds bounds checks to all array references

> Now just run all of the SPEC benchmarks through this preprocessor.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

> Compile them with your favorite compiler.

My favorite compiler doesn't handle any of the languages seen in the SPEC benchmarks, but it does automatically insert bounds checks for array references. Perhaps you meant, "favorite C compiler" :-).

> And then publish the "safe" versus "unsafe" benchmark times.

Ugh. I would rather you did not phrase it like that. The reason is that there are languages designed to have a "safe" implementation, and they run much more quickly than C-made-safe. This is not because of differences in the optimization algorithms, but because the design of the language (and the corresponding changes to calling conventions and data layouts) allows much more efficient implementation of checking. Bad results for "safe C" might be misinterpreted (might hell, they would, by those people who were so inclined to interpret them that way in the first place) to mean that safe languages in general are slow. This need not be the case.

Furthermore, there is some semantic information available in these languages that is not available in C or C++. If I see something like

```
for (i = 0; i < a.length; i++) {
  ... a[i] ...
}
```

you have what a friend of mine has called a "road kill bounds check". There's versions of this for Eiffel, Modula-3, and Ada, but not for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C. There are cases in C where a static array bound is supplied, but such code is relatively rare (because it is so inflexible, unlike Java arrays or Modula-3 open arrays) and you often still have to trust the cast to that type (not always, but sometimes). I worked on a C/C++-checking-product in a former job, and implementing the standard set of checks for C/C++, that come at very low cost in Modula-3 (a compiled language that I also work in from time to time), is very, very costly, especially in an environment where not all code is subject to those checks and binary compatibility with "unsafe" code (other people's libraries) is required. A 10x slowdown was our wildest-dreams design goal when we set out to build it.

There are other problems, too -- differences in the type system can affect the cost of performing some of the run-time-type checks, or it can affect the number of places where the type must be checked.

> If nothing else this would encourage compilers to eliminate unnecessary
> bounds checks...

Maybe, but it's a really hard way to do it. I'd rather spend my time optimizing Java bytecodes.

David Chase

A.3 Zalman Stern

Newsgroups: comp.arch

From: zalman@netcom.com (Zalman Stern)

Subject: Re: Naive Java question: Array index checking

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Date: Wed, 23 Apr 1997 00:37:08 GMT

Andy Glew (glew@cs.wisc.edu) wrote:

: Will somebody please write a C language to C language preprocessor

: that adds bounds checks to all array references, and which, if they

: fail, simply branches to a

:

: fprintf(stderr,

: "Array bounds check number %d failed\n",

: check_number);

: exit(1);

: code fragment.

: Now just run all of the SPEC benchmarks through this preprocessor.

: Compile them with your favorite compiler.

: And then publish the "safe" versus "unsafe" benchmark times.

: If nothing else this would encourage compilers to eliminate unnecessary

: bounds checks...

Hmmmmm... I suspect a Devil's Advocate position here...

Surely Andy is familiar with the Safe-C work covered in:

http://www.cs.wisc.edu/~austin/talk.scc/

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

(For those who do not get the "surely," this work was done by Todd Austin and Scott Breach under the guidance of Gurindar Sohi. I believe Dr. Austin has since gone to work for Andy's recent alma mater in the wilds of the Pacific Northwest. Dr. Sohi's research group is mentioned on Andy's home page. But hey, I've forgotten about plenty of code I've written, much less stuff that was done in my near midst...)

The above URL will get you to a paper which describes how to detect *all* spatial and temporal memory access errors in almost any C program. (Programs which depend on casting between integers and pointers will need to be adorned slightly however.) Its way cool and I'm pretty sure I've posted about it in comp.arch before...

The research was done using a source to source (C to C++?) preprocessor for C. I don't think anything has been done with SafeC since as it was a class project in a graduate level compilers course and the authors did/are doing something else for their theses. (So the conspiracy against array bounds checking runs so deep that one can't even get a thesis out of it anymore :-))

Before we start talking about adding hardware support for a feature involving runtime internals, we must very carefully design at least one, if not many, implementations of the runtime. For example, the Austin/Breach/Sohi work as presented in the paper above would not use an instruction that took three registers and signaled an exception if a value in "the middle" register was not within the bounds of the two "outer" registers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Using a relatively naive preprocessor approach, SafeC is 2.3x to 6.4x
slower (130% to 540% overhead) than the unadorned C code on a series of six
benchmarks. Programs are typically less than twice as large with checks.
(More detailed data is given in the paper, including breakdowns of where
the time and space goes.) The checking provide is significantly more
extensive than what most people mean when they say "array bounds checking."
They also give some lower bounds on how well an optimizer that elides
unnecessary checks can do.

Certainly if I were in "the hardware community" I would not believe the
first software person who said "The world will end in a flurry of lawsuits
if you support free array bounds checking." But if I did I would start by
asking questions like "What do you mean by free?" "What exactly must be
checked?" And on down the line.

How would a hardware designer feel about adding lots of hardware support
for fast array bounds checking to have something like a better version of
SafeC make it all moot?

-Z-

A.4 David Chase

Newsgroups: comp.arch
From: hbaker@netcom.com (Henry Baker)
Subject: Re: Naive Java question: Array index checking
Date: Wed, 23 Apr 1997 17:08:08 GMT

In article <ygezpuqvxaz.fsf@algor.doc.ic.ac.uk>, jsc@doc.ic.ac.uk (Stephen

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Crane) wrote:

> I see bounds-checking as a debugging utility: programmer writes
> code and tests it. During testing it breaks due to memory corruption.
> Programmer turns on bounds-checking, recompiles and hey presto, finds
> the bug and fixes it. Iterate. Program working, programmer
> recompiles with bounds-checking disabled.

I hope that this a HW and not a SW person suggesting this.

Unfortunately, for any sufficiently large and complex program, the 'testing' phase is _never_ done. This is the result of our inability to solve the halting problem.

Since one can never be sure that all bugs have been vanquished, you now have to decide what to do about the bugs that remain. The head-in-the-sand 'what, me worry?' attitude seems to be the preference of many software organizations. They turn off all checking, and pray that any failures will be so violent that the program will crash completely rather than leaving subtle corruptions in the database.

The more enlightened organizations admit, up-front, that potential bugs remain, and provide for detecting them before they can corrupt important data, and supply enough information in the form of error messages so that the bug can be killed at the source. More enlightened customers will be grateful for this, since they have been protected from the worst consequences of bugs that do show themselves.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Since the existence of bugs can also enable significant security violations in today's networks, the lack of such checks could eventually open up the software vendor to legal problems.

A.5 Henry Baker

Newsgroups: comp.arch

From: hbaker@netcom.com (Henry Baker)

Subject: Re: Naive Java question: Array index checking

In article <ygezpuqvxaz.fsf@algor.doc.ic.ac.uk>, jsc@doc.ic.ac.uk (Stephen Crane) wrote:

> I see bounds-checking as a debugging utility: programmer writes
> code and tests it. During testing it breaks due to memory corruption.
> Programmer turns on bounds-checking, recompiles and hey presto, finds
> the bug and fixes it. Iterate. Program working, programmer
> recompiles with bounds-checking disabled.

I hope that this a HW and not a SW person suggesting this.

Unfortunately, for any sufficiently large and complex program, the 'testing' phase is _never_ done. This is the result of our inability to solve the halting problem.

Since one can never be sure that all bugs have been vanquished, you now have to decide what to do about the bugs that remain. The head-in-the-sand 'what, me worry?' attitude seems to be the preference of many software organizations. They turn off all checking, and pray that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

any failures will be so violent that the program will crash completely rather than leaving subtle corruptions in the database.

The more enlightened organizations admit, up-front, that potential bugs remain, and provide for detecting them before they can corrupt important data, and supply enough information in the form of error messages so that the bug can be killed at the source. More enlightened customers will be grateful for this, since they have been protected from the worst consequences of bugs that do show themselves.

Since the existence of bugs can also enable significant security violations in today's networks, the lack of such checks could eventually open up the software vendor to legal problems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Appendix B

Additional Data for Purify Experiments

Section B.1 describes additional errors reported by Purify. Section B.2 presents the complete list of programs tested. Section B.3 presents the raw data of access errors discovered for this research.

B.1 Additional Errors

Only five types of errors were presented in the body of this work. Purify actually reports seven types of errors. The two additional types are given in the following two paragraphs.

Freeing unallocated memory error (fum). Memory was being freed which was already in the heap. Usually this indicates a redundant free operation to a previously freed memory block.

Freeing non-heap error (fnh). Memory which was never part of the heap was being freed. This is a serious programming error usually indicative of a trashed memory block pointer.

These two errors are not necessarily access errors but more memory use errors. Both are errors in freeing memory.

B.2 Programs

Table 2 lists the thirty one programs and indicates which package each belongs to as well as the programming language used in each.

Program	Package	Language
GateToGate	MCC CCS-2.5	C++

Table 2: Programs Tested

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Program	Package	Language
XCellCompact	MCC CCS-2.5	C++
XCellPlacer	MCC CCS-2.5	C++
XCellRouter	MCC CCS-2.5	C++
placement	MCC CCS-2.5	C
router	MCC CCS-2.5	C
zcat	GNU gzip-1.2.4	C
tar	GNU tar-1.11.2	C
make	GNU make-3.70	C
sed	GNU sed-2.03	C
gawk	GNU gawk-2.15.4	C
gcc	GNU gcc-2.5.8	C
cc1	GNU gcc-2.5.8	C
ccp	GNU gcc-2.5.8	C
geqn	GNU groff-1.01	C++
gpic	GNU groff-1.01	C++
grodvi	GNU groff-1.01	C++
groff	GNU groff-1.01	C++
gsoelim	GNU groff-1.01	C++
gtbl	GNU groff-1.01	C++
gtroff	GNU groff-1.01	C++
doc	InverViews-3.1 Beta	C++
ibuild	InverViews-3.1 Beta	C++
idraw	InverViews-3.1 Beta	C++
iclass	InverViews-3.1 Beta	C++
ghostscript	GNU ghostscript-2.61	C
temacs	GNU emacs-18.59	C
ldl++	MCC Carnot-2	C++
amd	BSD amd5.3-beta1	C

Table 2: Programs Tested (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Program	Package	Language
pplc	parprosys	C/C++
sendmail	BSD sendmail-8.6.4	C

Table 2: Programs Tested (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

B.3 DATA

B.3.1 Dynamic data

program	umr	abr	abw	fmr	fmw	fnh	fum
GateToGate	2	0	0	0	0	0	0
XCellCompact	0	0	0	0	0	0	0
XCellPlacer	0	4	0	61	0	1	0
XCellRouter	0	0	0	0	0	0	0
placement	107	0	0	3	0	0	0
router	298	4552	51	116	0	0	0
zcat	0	0	0	0	0	0	0
tar	0	0	0	0	0	0	0
make	1207	0	0	0	0	0	0
sed	0	0	0	0	0	0	0
gawk	6	0	0	0	0	0	0
gcc	0	0	0	0	0	0	0
cc1	1781	0	0	0	0	0	0
ccp	4806	0	0	0	0	0	0
geqn	0	0	0	0	0	0	0
gpic	0	0	0	0	0	0	0
grodvi	1	0	0	0	0	0	0
groff	0	0	0	0	0	0	0
gsoelim	0	0	0	0	0	0	0
gtbl	0	0	0	0	0	0	0
gtroff	2108	0	0	0	0	0	0
doc	2	0	0	0	0	0	0
ibuild	2832	0	0	63	5	0	5
idraw	3	0	0	0	0	0	0
iclass	3	9	0	0	0	0	0

Table 3: Dynamically reported errors.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

program	umr	abr	abw	fmr	fmw	fnh	fum
ghostscript	3571	317	317	0	0	0	0
temacs	0	0	0	0	0	0	0
ldl++	0	0	0	2156	397	0	52
amd	20	0	0	1	1	0	0
pplc	6	505	223	0	0	0	0
sendmail	42	0	0	0	0	0	0

Table 3: Dynamically reported errors. (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

B.3.2 Static data

program	umr	abr	abw	fmr	fmw	fnh	fum
GateToGate	2	0	0	0	0	0	0
XCellCom-pact	0	0	0	0	0	0	0
XCellPlacer	0	2	0	2	0	1	0
XCellRouter	0	0	0	0	0	0	0
placement	13	0	0	2	0	0	0
router	11	11	2	17	0	0	0
zcat	0	0	0	0	0	0	0
tar	0	0	0	0	0	0	0
make	3	0	0	0	0	0	0
sed	0	0	0	0	0	0	0
gawk	2	0	0	0	0	0	0
gcc	0	0	0	0	0	0	0
cc1	2	0	0	0	0	0	0
ccp	1	0	0	0	0	0	0
geqn	0	0	0	0	0	0	0
gpic	0	0	0	0	0	0	0
grodvi	1	0	0	0	0	0	0
groff	0	0	0	0	0	0	0
gsoelim	0	0	0	0	0	0	0
gtbl	0	0	0	0	0	0	0
gtroff	2	0	0	0	0	0	0
doc	2	0	0	0	0	0	0
ibuild	9	0	0	12	1	0	1
idraw	1	0	0	0	0	0	0
iclass	1	1	0	0	0	0	0
ghostscript	5	1	1	0	0	0	0

Table 4: Source code statements generating errors.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

program	umr	abr	abw	fmr	fmw	fnh	fum
temacs	0	0	0	0	0	0	0
idl++	0	0	0	38	20	0	1
amd	4	0	0	1	1	0	0
pplc	4	25	8	0	0	0	0
sendmail	1	0	0	0	0	0	0

Table 4: Source code statements generating errors. (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Appendix C

XSim Experimental Data

C.1 Experiment One

C.1.1 Experiment One Part One Data

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Cycles	19475537	19680914	548610	5899339	23031893	38795786
Instructions	23536536	23638300	514187	6834912	22269392	41253134
IPC	1.208518	1.201077	0.937254	1.158589	0.966894	1.063341
Two Inst	6445524	6482275	147716	2242592	6943089	12980788
One Inst	10645488	10673750	218755	2349728	8383214	15291558
Zero Inst	2384525	2524889	182139	1307019	7705590	10523440
No Inst	2313294	2417598	164940	775677	6381275	9352633
Source Unav	8490211	8547341	124209	1633037	6126289	9476997
Destination Unav	16048	12261	3901	3067	210354	83883
Pipe Full	0	0	0	0	0	0
Reservation Full	5554	10783	4306	3991	150060	939041
DMU B/W	2148655	2149391	67076	621145	1999253	2654324
Empty D Slot	35125	34482	25563	349725	598703	1329520
Branch-Branch	14044	17435	8963	264695	504182	1912772
History Full	46	91	7	319	8087	581
Serialized	5125	5332	208	4863	7439	60900
Carry bit	0	0	0	0	305	4
Decode Error	0	0	0	0	0	0
	1	1	1	1	1	1
Integer 1	10650611	10698084	193203	2361583	8713230	15246840
Integer 1 %	45.3	45.3	37.6	34.6	39.1	37

Table 5: Experimental Data One Part One

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
	2	2	2	2	2	2
Integer 2	2105318	2104591	14686	273728	896803	1151598
Integer 2 %	8.9	8.9	2.9	4	4	2.8
Load	6372353	6381816	124601	1514788	4831271	8881216
Load %	27.1	27	24.2	22.2	21.7	21.5
Store	2155998	2182008	62313	894573	2561612	4952805
Store %	9.2	9.2	12.1	13.1	11.5	12
Float Add	0	0	0	0	32849	0
Float Add %	0	0	0	0	0.1	0
Float Mul	4622	2776	2475	0	59956	2758
Float Mul %	0	0	0.5	0	0.3	0
Float Div	162	791	0	12	34147	18237
Float Div %	0	0	0	0	0.2	0
Graph Add	0	0	0	0	0	0
Graph Add %	0	0	0	0	0	0
Graph Bit	0	0	0	0	0	0
Graph Bit %	0	0	0	0	0	0
Cond Branch	2195602	2200518	91876	1246191	3480480	8099126
Cond Branch %	9.3	9.3	17.9	18.2	15.6	19.6
Uncond Branch	49205	65047	24930	541595	1655315	2870105
Uncond Branch %	0.2	0.3	4.8	7.9	7.4	7
Trap	2663	2667	101	2440	3727	30447
Trap %	0	0	0	0	0	0.1
Rte	0	0	0	0	0	0
Rte %	0	0	0	0	0	0
Control	2	2	2	2	2	2
Control %	0	0	0	0	0	0
Load Latency	2.68	2.7	2.55	2.68	2.89	2.51

Table 5: Experimental Data One Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Load Clocks	2.01	2.01	1.98	2.01	2.21	2.03
Load hits	0	0	0	0	0	0
Decoupled Loads	1870	3538	1024	1173	33041	27040
Decoupled Stores	1840	2084	33	1393	4226	8555
address alias	219	108	59	171	14803	29973
ld input full	245	28	174	161	5158	2348
ld output full	0	0	0	0	0	0
st reservation full	5309	10755	4132	3830	144902	936693
Cond store	43	75	48	0	9075	2163
Inst/Branch	10.48	10.43	4.4	3.82	4.34	3.76
BTC hits	28888	23228	16414	522032	930043	2873302
BTC misses	10827	20244	9197	3866	324703	524438
Correct Pred	46568	42759	38074	618899	1481167	3921042
Incorrect Pred	25105	24303	15806	96483	746751	1275547
History Depth	0.66	0.57	0.9	2.68	1.25	1.44
Flush History	13646	11273	11356	173754	678884	1117848
Flush Stall	0	0	0	0	0	0
Accesses	14993831	15051032	347713	4945717	15498033	27657518
Hits	14971514	15013402	332678	4941710	15027284	27213393
Stream Hits	5420	9282	3314	1036	136942	109925
Misses	16897	28348	11721	2971	333807	334200
Ratio	99.89	99.81	96.63	99.94	97.85	98.79
Accesses	8525124	8561580	183190	2239325	7081693	13481941
Hits	8513110	8541760	181643	2236310	6930228	13393712
Stream Hits	19	53	12	7	1848	1896
Misses	11995	19767	1535	3008	149617	86333
Copybacks	3018	6421	807	2139	49275	38002
Ratio	99.86	99.77	99.16	99.87	97.89	99.36

Table 5: Experimental Data One Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Cycles used	251181	416776	115588	58646	4097565	3467495
Conflicts	1280	1563	825	425	48379	22671
Bus Utilization	0.012897	0.021177	0.210692	0.009941	0.177908	0.089378
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
exit	1	1	1	1	1	1
read	2154	2150	7	30	23	55
write	0	6	85	2381	3696	30379
open	84	84	1	5	0	2
close	84	85	1	5	0	1
brk	2	2	2	4	2	2
time	0	0	0	0	0	0
times	0	0	0	0	0	0
sysconf	0	0	0	0	0	0
ioctl	83	83	0	7	3	4
lseek	0	0	0	0	0	0
fstat	83	83	0	7	2	3
getpid	0	0	0	0	0	0
fcntl	4	4	4	0	0	0
access	0	0	0	0	0	0
creat	0	1	0	0	0	0
unlink	0	0	0	0	0	0
stat	0	0	0	0	0	0
lstat	168	168	0	0	0	0

Table 5: Experimental Data One Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Updates	19289	29497	11633	181889	732805	837389
Bytes Allocated	1016128	1214384	483616	7260032	26941520	57495456
Lines Allocated	33272	41729	16380	270579	947796	1952873
Bytes Deallocated	1016080	1214336	483568	7259984	26941472	57495408
Lines Deallocated	33270	41727	16378	270577	947794	1952871
Max Depth	3776	3952	2224	2112	2912	2160
Max Lines	121	126	73	68	96	70
Mallocs	439	603	187	393	4857	6834
Bytes Malloc	355494	378215	9859	33482	251743	150742
Lines Malloc	11312	12025	382	1293	10400	8596
Frees	255	589	2	331	4863	6840
Jsr/Bsr	24691	31970	13790	95596	595321	1081526
Jmp r1	18537	27531	7586	94575	529347	644995
Leaf Nodes	11054	17583	3445	87786	329555	449941

Table 5: Experimental Data One Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.1.2 Experiment One Part Two Data

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Cycles	16005741	306050485	55478409	11700482	97089967	11686619
Instructions	16836344	243693938	52607713	6465443	72045628	6471169
IPC	1.051894	0.796254	0.948256	0.552579	0.74205	0.553725
Two Inst	5265708	72753084	16184410	1955556	21455024	1908477
One Inst	6304928	98187770	20238893	2554331	29135580	2654215
Zero Inst	4435105	135109631	19055106	7190595	46499363	7123927
No Inst	4088414	101322557	13312481	1679592	8899440	1725214
Source Unav	3948374	80383265	17818462	7296798	44927328	7306131
Destination Unav	63040	4483951	386514	37210	6427815	40696
Pipe Full	0	0	0	0	0	0
Reservation Full	191559	4370135	429739	90837	350191	87665
DMU B/W	1128462	26801094	4521276	358059	6474022	357875
Empty D Slot	652782	7368020	1231201	183967	899181	183169
Branch-Branch	641573	8137097	1427477	54701	655443	31345
History Full	1194	27684	533	902	0	234
Serialized	16132	67683	56152	15535	3100194	18004
Carry bit	2240	4480	1	0	0	0
Decode Error	0	0	0	0	0	0
	1	1	1	1	1	1
Integer 1	6113384	86666260	19559504	2644251	20512684	2667583
Integer 1 %	36.3	35.6	37.2	40.9	28.5	41.2
	2	2	2	2	2	2
Integer 2	677137	7118782	1403668	195198	3878681	176698
Integer 2 %	4	2.9	2.7	3	5.4	2.7
Load	3460349	60490350	12711447	722113	14079364	715394
Load %	20.6	24.8	24.2	11.2	19.5	11.1
Store	2201527	30098096	6119773	358988	6275375	355868

Table 6: Experimental Data One Part Two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Store %	13.1	12.4	11.6	5.6	8.7	5.5
Float Add	0	95739	4	207713	11801319	214407
Float Add %	0	0	0	3.2	16.4	3.3
Float Mul	4706	486537	19355	796831	5317973	795374
Float Mul %	0	0.2	0	12.3	7.4	12.3
Float Div	1246	757367	139028	274603	570988	274317
Float Div %	0	0.3	0.3	4.2	0.8	4.2
Graph Add	0	0	0	0	0	0
Graph Add %	0	0	0	0	0	0
Graph Bit	0	0	0	0	0	0
Graph Bit %	0	0	0	0	0	0
Cond Branch	3012805	36685263	8969154	1018370	5314223	1017416
Cond Branch %	17.9	15.1	17	15.8	7.4	15.7
Uncond Branch	1357118	21261679	3657698	242833	3444961	248616
Uncond Branch %	8.1	8.7	7	3.8	4.8	3.8
Trap	8070	33863	28080	1721	250022	1990
Trap %	0	0	0.1	0	0.3	0
Rte	0	0	0	0	0	0
Rte %	0	0	0	0	0	0
Control	2	2	2	2822	600038	3506
Control %	0	0	0	0	0.8	0.1
Load Latency	2.49	3.09	2.68	2.36	2.7	2.37
Load Clocks	1.98	2.22	2.09	1.93	2.28	1.96
Load hits	1	89	1	0	0	0
Decoupled Loads	23795	632047	22054	2451	75	2280
Decoupled Stores	1902	131372	9883	52	8	45
address alias	11348	96035	29105	8452	3699494	9310
ld input full	2630	418023	10134	100	50013	175

Table 6: Experimental Data One Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
ld output full	0	0	0	0	0	0
st reservation full	188929	3952112	419605	90737	300178	87490
Cond store	683	26568	777	251	32	504
Inst/Branch	3.85	4.21	4.17	5.13	8.23	5.11
BTC hits	767900	8273839	1959839	231624	1293409	212576
BTC misses	349474	5624873	837792	129272	327839	107569
Correct Pred	1493244	14382316	4284761	419264	2526341	440644
Incorrect Pred	510926	9029527	1835329	229948	1973388	235952
History Depth	1.13	0.68	0.94	1.05	1.97	1.04
Flush History	398147	4909484	1176959	141948	2610683	141827
Flush Stall	0	0	0	0	0	0
Accesses	11449352	170602407	36650791	4573349	50812134	4486232
Hits	11234518	161190694	35874679	4467893	50811549	4385177
Stream Hits	69855	2403975	203959	28290	151	30041
Misses	144979	7007738	572153	77166	434	71014
Ratio	98.73	95.89	98.44	98.31	100	98.42
Accesses	5477952	88043268	18403191	1056840	19413113	1048186
Hits	5442617	85243946	18154903	1051148	19412832	1041727
Stream Hits	1336	167405	807	256	3	226
Misses	33999	2631917	247481	5436	278	6233
Copybacks	23976	1129689	99850	3877	64	4135
Ratio	99.38	97.01	98.66	99.49	100	99.41
Cycles used	1524308	86292224	6836127	703681	5895	651603
Conflicts	13269	1065878	24774	1675	107	1743
Bus Utilization	0.095235	0.281954	0.123221	0.060141	0.000061	0.055756
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0

Table 6: Experimental Data One Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
exit	1	1	1	1	1	1
read	33	207	128	0	0	0
write	8027	33567	27880	539	0	523
open	0	28	24	0	0	0
close	0	15	11	0	0	0
brk	4	10	2	2	2	2
time	0	1	1	0	0	0
times	0	0	0	0	0	0
sysconf	0	0	0	0	0	0
ioctl	3	17	15	0	0	0
lseek	0	0	1	0	0	0
fstat	2	16	14	0	0	0
getpid	0	1	1	0	0	0
fcntl	0	0	0	4	4	4
access	0	0	1	0	0	0
creat	0	0	0	0	0	0
unlink	0	0	1	0	0	0
stat	0	0	0	0	0	0
lstat	0	0	0	0	0	0
Updates	541485	10955610	1628129	73617	1300713	78931
Bytes Allocated	24371792	457432656	64888848	5724832	48020800	5849792
Lines Allocated	869485	15871526	2282433	187998	1750761	193453
Bytes Deallocated	24371744	457432080	64888800	5724784	48020752	5849744
Lines Deallocated	869483	15871507	2282431	187996	1750759	193451
Max Depth	2000	6672	2944	6928	544	6960

Table 6: Experimental Data One Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Max Lines	65	212	95	220	19	222
Mallocs	8353	111434	2897	271	25	271
Bytes Malloc	101357	3393967	324299	54278	3018	36482
Lines Malloc	9493	171767	11230	1790	104	1257
Frees	8484	94622	2929	259	21	259
Jsr/Bsr	341061	8841575	1193526	84831	1578118	88562
Jmp r1	325130	7531027	1054925	57054	1328085	59394
Leaf Nodes	229405	4593390	710450	43285	1027836	43540

Table 6: Experimental Data One Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.1.3 Experiment One Part Three Data

Parameter	hsim	mpsdemo	z80sim	photon
Cycles	50278779	1434388	164915003	77018190
Instructions	51083648	1035196	168680934	66729117
IPC	1.016008	0.721699	1.022836	0.866407
Two Inst	15421157	250596	49402361	19041135
One Inst	20241334	534004	69876212	28646847
Zero Inst	14616288	649788	45636430	29330208
No Inst	8303010	555336	40118965	20809056
Source Unav	17449848	334947	38431225	19762438
Destination Unav	202775	26845	226287	1564476
Pipe Full	0	0	0	0
Reservation Full	202263	8734	1579908	1770988
DMU B/W	6729248	188913	29597510	9450770
Empty D Slot	1261569	59290	5084051	2794300
Branch-Branch	705962	7759	394718	968313
History Full	0	48	0	5
Serialized	968	913	16	412673
Carry bit	0	0	0	0
Decode Error	0	0	0	0
	1	1	1	1
Integer 1	18035714	350688	60896769	24970887
Integer 1 %	35.3	33.9	36.1	37.4
	2	2	2	2
Integer 2	1840245	31266	7040142	2583410
Integer 2 %	3.6	3	4.2	3.9
Load	14533102	328950	45227784	14940702
Load %	28.4	31.8	26.8	22.4
Store	4094789	151390	25616181	9598358

Table 7: Experimental Data One Part Three

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Store %	8	14.6	15.2	14.4
Float Add	0	0	0	819012
Float Add %	0	0	0	1.2
Float Mul	953352	476	0	933456
Float Mul %	1.9	0	0	1.4
Float Div	201380	283	0	355089
Float Div %	0.4	0	0	0.5
Graph Add	0	0	0	0
Graph Add %	0	0	0	0
Graph Bit	0	0	0	0
Graph Bit %	0	0	0	0
Cond Branch	8593194	91109	11016342	7661540
Cond Branch %	16.8	8.8	6.5	11.5
Uncond Branch	2830918	80566	18883707	4743389
Uncond Branch %	5.5	7.8	11.2	7.1
Trap	952	466	7	36872
Trap %	0	0	0	0.1
Rte	0	0	0	0
Rte %	0	0	0	0
Control	2	2	2	86402
Control %	0	0	0	0.1
Load Latency	2.2	2.6	2.35	2.54
Load Clocks	1.91	2.05	1.81	1.95
Load hits	0	1	0	0
Decoupled Loads	70	1350	18496	89794
Decoupled Stores	12	84	4	1838
address alias	44	11618	202886	462922
ld input full	11	44	0	20220

Table 7: Experimental Data One Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
ld output full	0	0	0	0
st reservation full	202252	8690	1579908	1750768
Cond store	66	432	0	12666
Inst/Branch	4.47	6.03	5.64	5.38
BTC hits	4482823	21191	1945204	1734687
BTC misses	2015	58056	1196668	3113527
Correct Pred	4430702	46774	5672435	3396861
Incorrect Pred	1694922	26160	400522	1450636
History Depth	1.25	0.92	0.46	0.96
Flush History	1313908	16538	185385	1059757
Flush Stall	0	0	0	0
Accesses	35504441	704740	122725551	47838592
Hits	35499272	639899	120328223	45818533
Stream Hits	1962	12248	491494	512682
Misses	3207	52593	1905834	1507377
Ratio	99.99	92.54	98.45	96.85
Accesses	17719803	472417	70825524	24099189
Hits	17719564	467025	70806364	23943189
Stream Hits	1	109	5	1678
Misses	238	5283	19155	154322
Copybacks	25	2753	6573	66812
Ratio	100	98.88	99.97	99.36
Cycles used	24933	498069	15569111	13511099
Conflicts	60	2863	9907	35373
Bus Utilization	0.000496	0.347235	0.094407	0.175427
Accesses	0	0	0	0
Misses	0	0	0	0
Ratio	0	0	0	0

Table 7: Experimental Data One Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Accesses	0	0	0	0
Misses	0	0	0	0
Ratio	0	0	0	0
exit	1	1	1	1
read	936	14	0	8
write	7	433	0	849
open	1	6	0	4
close	0	6	0	4
brk	2	2	2	2
time	0	0	0	0
times	0	0	0	0
sysconf	0	0	0	0
ioctl	0	0	0	0
lseek	0	0	0	0
fstat	0	0	0	0
getpid	0	0	0	0
fcntl	5	4	4	4
access	0	0	0	0
creat	0	0	0	0
unlink	0	0	0	0
stat	0	0	0	0
lstat	0	0	0	0
Updates	1616353	75330	9008479	2966840
Bytes Allocated	51728576	1615248	548799536	130316512
Lines Allocated	1718440	58887	18022579	4313879
Bytes Deallocated	51728528	1615200	548799488	130316464
Lines Deallocated	1718438	58885	18022577	4313877

Table 7: Experimental Data One Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Max Depth	560	1920	784	32448
Max Lines	19	65	27	1017
Mallocs	13	173	10	3730
Bytes Malloc	5280	27580	68221	799360
Lines Malloc	167	930	2133	25160
Frees	0	138	0	4
Jsr/Bsr	1112760	33367	6833472	1713932
Jmp r1	1011636	27570	6833468	1453952
Leaf Nodes	505713	15241	4805653	1112470

Table 7: Experimental Data One Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.2 Experiment Two Data

C.2.1 Experiment Two Part One Data

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Cycles	19511414	19738853	565105	5907705	23610691	39290174
Instructions	23536534	23638330	514187	6834912	22264991	41253187
IPC	1.206296	1.197553	0.909896	1.156949	0.943005	1.049962
Two Inst	6445497	6482237	147711	2242591	6941378	12980886
One Inst	10645540	10673856	218765	2349730	8382235	15291415
Zero Inst	2420377	2582760	198629	1315384	8287078	11017873
No Inst	2335623	2452540	179562	779368	6797799	9739786
Source Unav	8500794	8568048	125256	1636584	6250138	9555989
Destination Unav	18131	13058	4227	3491	232259	92275
Pipe Full	0	0	0	0	0	0
Reservation Full	6340	11965	4829	4447	163743	956382
DMU B/W	2148550	2149335	66992	621149	1998237	2654610
Empty D Slot	35107	34466	25563	349724	598615	1329498
Branch-Branch	14269	17760	9024	264713	508384	1913932
History Full	74	129	9	537	9283	1336
Serialized	5131	5335	209	4862	7438	60901
Carry bit	0	0	0	0	308	5
Decode Error	0	0	0	0	0	0
	1	1	1	1	1	1
Integer 1	10650616	10698071	193203	2361583	8711837	15246791
Integer 1 %	45.3	45.3	37.6	34.6	39.1	37
	2	2	2	2	2	2
Integer 2	2105314	2104606	14686	273728	896380	1151721
Integer 2 %	8.9	8.9	2.9	4	4	2.8

Table 8: Experimental Data Two Part One

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Load	6372347	6381817	124601	1514788	4830372	8881305
Load %	27.1	27	24.2	22.2	21.7	21.5
Store	2155998	2182008	62313	894573	2561082	4952775
Store %	9.2	9.2	12.1	13.1	11.5	12
Float Add	0	0	0	0	32849	0
Float Add %	0	0	0	0	0.1	0
Float Mul	4622	2776	2475	0	59956	2758
Float Mul %	0	0	0.5	0	0.3	0
Float Div	162	791	0	12	34147	18237
Float Div %	0	0	0	0	0.2	0
Graph Add	0	0	0	0	0	0
Graph Add %	0	0	0	0	0	0
Graph Bit	0	0	0	0	0	0
Graph Bit %	0	0	0	0	0	0
Cond Branch	2195602	2200518	91876	1246191	3479676	8099126
Cond Branch %	9.3	9.3	17.9	18.2	15.6	19.6
Uncond Branch	49208	65074	24930	541595	1654967	2870025
Uncond Branch %	0.2	0.3	4.8	7.9	7.4	7
Trap	2663	2667	101	2440	3723	30447
Trap %	0	0	0	0	0	0.1
Rte	0	0	0	0	0	0
Rte %	0	0	0	0	0	0
Control	2	2	2	2	2	2
Control %	0	0	0	0	0	0
Load Latency	2.68	2.71	2.58	2.68	2.98	2.54
Load Clocks	2.01	2.02	2	2.02	2.25	2.04
Load hits	0	0	0	0	0	0
Decoupled Loads	1887	3900	1046	1302	36881	30523

Table 8: Experimental Data Two Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Decoupled Stores	1846	2100	35	1397	4688	8622
address alias	278	141	70	216	15898	31617
ld input full	350	23	191	184	6161	2895
ld output full	0	0	0	0	0	0
st reservation full	5990	11942	4638	4263	157582	953487
Cond store	68	89	69	0	12210	3195
Inst/Branch	10.48	10.43	4.4	3.82	4.34	3.76
BTC hits	28868	23212	16414	522032	929443	2873123
BTC misses	10839	20270	9197	3866	324714	524608
Correct Pred	46539	42755	38089	618898	1480763	3921208
Incorrect Pred	25107	24361	15806	96483	746749	1275693
History Depth	0.66	0.57	0.9	2.68	1.25	1.44
Flush History	13653	11300	11356	173754	678798	1117761
Flush Stall	0	0	0	0	0	0
Accesses	14993775	15051193	347636	4945784	15495013	27658688
Hits	14971384	15013610	332637	4941777	15023614	27214422
Stream Hits	5458	9195	3314	1036	136820	110314
Misses	16933	28388	11685	2971	334579	333952
Ratio	99.89	99.81	96.64	99.94	97.84	98.79
Accesses	8525113	8561576	183190	2239325	7080051	13481986
Hits	8513150	8541704	181642	2236310	6929089	13393649
Stream Hits	20	59	13	7	1810	1911
Misses	11943	19813	1535	3008	149152	86426
Copybacks	3004	6370	807	2139	49130	38039
Ratio	99.86	99.77	99.16	99.87	97.89	99.36
Cycles used	287289	477198	132330	67384	4693655	3971867
Conflicts	1411	1664	828	431	49079	23046
Bus Utilization	0.014724	0.024176	0.234169	0.011406	0.198794	0.101091

Table 8: Experimental Data Two Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
exit	1	1	1	1	1	1
read	2154	2150	7	30	23	55
write	0	6	85	2381	3692	30379
open	84	84	1	5	0	2
close	84	85	1	5	0	1
brk	2	2	2	4	2	2
time	0	0	0	0	0	0
times	0	0	0	0	0	0
sysconf	0	0	0	0	0	0
ioctl	83	83	0	7	3	4
lseek	0	0	0	0	0	0
fstat	83	83	0	7	2	3
getpid	0	0	0	0	0	0
fcntl	4	4	4	0	0	0
access	0	0	0	0	0	0
creat	0	1	0	0	0	0
unlink	0	0	0	0	0	0
stat	0	0	0	0	0	0
lstat	168	168	0	0	0	0
Updates	19287	29493	11633	181889	732679	837375
Bytes Allocated	1016080	1214288	483616	7260032	26934976	57495376
Lines Allocated	33270	41725	16380	270579	947548	1952867

Table 8: Experimental Data Two Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Bytes Deallocated	1016032	1214240	483568	7259984	26934928	57495328
Lines Deallocated	33268	41723	16378	270577	947546	1952865
Max Depth	3776	3952	2224	2112	2912	2160
Max Lines	121	126	73	68	96	70
Mallocs	439	603	187	393	4857	6834
Bytes Malloc	355494	378215	9859	33482	251743	150742
Lines Malloc	11312	12025	382	1293	10400	8596
Frees	255	589	2	331	4858	6839
Jsr/Bsr	24694	31988	13790	95596	595144	1081521
Jmp r1	18537	27540	7586	94575	529333	644900
Leaf Nodes	11053	17603	3445	87786	329485	449941

Table 8: Experimental Data Two Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.2.2 Experiment Two Part Two Data

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Cycles	16219988	318438167	56459296	11799065	97090701	11779053
Instructions	16836453	243691616	52607537	6465414	72045628	6471129
IPC	1.038007	0.765271	0.931778	0.54796	0.742045	0.549376
Two Inst	5265546	72743261	16184343	1955520	21455020	1908513
One Inst	6305361	98205094	20238851	2554374	29135588	2654103
Zero Inst	4649081	147489812	20036102	7289171	46500093	7216437
No Inst	4257683	110455185	13965623	1772308	8900004	1810580
Source Unav	3977585	82721875	18083915	7299587	44927433	7309809
Destination Unav	71793	4931281	394291	37664	6427809	41756
Pipe Full	0	0	0	0	0	0
Reservation Full	196867	4776065	460341	93217	350232	89893
DMU B/W	1128537	26800997	4520544	358087	6474022	357863
Empty D Slot	652729	7367322	1231202	183973	899181	183100
Branch-Branch	642099	8198350	1428333	54774	655476	31427
History Full	2074	38983	945	1052	0	288
Serialized	16135	67695	56130	15535	3100194	17992
Carry bit	2249	4492	1	0	0	0
Decode Error	0	0	0	0	0	0
	1	1	1	1	1	1
Integer 1	6113326	86665890	19559382	2644243	20512685	2667562
Integer 1 %	36.3	35.6	37.2	40.9	28.5	41.2
	2	2	2	2	2	2
Integer 2	677233	7120183	1403794	195194	3878680	176690
Integer 2 %	4	2.9	2.7	3	5.4	2.7
Load	3460423	60489293	12711266	722114	14079364	715368
Load %	20.6	24.8	24.2	11.2	19.5	11.1
Store	2201524	30095059	6119781	358988	6275375	355881

Table 9: Experimental Data Two Part Two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Store %	13.1	12.3	11.6	5.6	8.7	5.5
Float Add	0	95739	4	207708	11801319	214409
Float Add %	0	0	0	3.2	16.4	3.3
Float Mul	4706	486537	19355	796831	5317973	795374
Float Mul %	0	0.2	0	12.3	7.4	12.3
Float Div	1246	757367	139028	274603	570988	274317
Float Div %	0	0.3	0.3	4.2	0.8	4.2
Graph Add	0	0	0	0	0	0
Graph Add %	0	0	0	0	0	0
Graph Bit	0	0	0	0	0	0
Graph Bit %	0	0	0	0	0	0
Cond Branch	3012805	36685263	8969156	1018370	5314223	1017416
Cond Branch %	17.9	15.1	17	15.8	7.4	15.7
Uncond Branch	1357118	21262420	3657689	242820	3444961	248616
Uncond Branch %	8.1	8.7	7	3.8	4.8	3.8
Trap	8070	33863	28080	1721	250022	1990
Trap %	0	0	0.1	0	0.3	0
Rte	0	0	0	0	0	0
Rte %	0	0	0	0	0	0
Control	2	2	2	2822	600038	3506
Control %	0	0	0	0	0.8	0.1
Load Latency	2.52	3.23	2.75	2.38	2.7	2.39
Load Clocks	2	2.28	2.11	1.94	2.28	1.97
Load hits	1	103	1	0	0	0
Decoupled Loads	25910	709175	23701	2908	84	2733
Decoupled Stores	1916	134120	9813	52	7	46
address alias	11973	117000	29706	8606	3699496	9740
ld input full	3307	493987	12766	119	50016	211

Table 9: Experimental Data Two Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
ld output full	0	0	0	0	0	0
st reservation full	193560	4282078	447575	93098	300216	89682
Cond store	834	36931	1089	273	52	525
Inst/Branch	3.85	4.21	4.17	5.13	8.23	5.11
BTC hits	767911	8275296	1960452	231464	1293409	212808
BTC misses	349420	5624247	837131	129429	327839	107319
Correct Pred	1493340	14384086	4284878	419243	2526334	440743
Incorrect Pred	510969	9029424	1835304	229949	1973388	235970
History Depth	1.13	0.68	0.94	1.05	1.97	1.04
Flush History	398219	4909465	1176809	141931	2610683	141785
Flush Stall	0	0	0	0	0	0
Accesses	11449994	170619111	36649435	4573494	50812148	4485360
Hits	11234815	161207221	35872809	4468041	50811563	4383931
Stream Hits	70448	2402516	204253	28288	151	30205
Misses	144731	7009374	572373	77165	434	71224
Ratio	98.74	95.89	98.44	98.31	100	98.41
Accesses	5477910	88042859	18403255	1056840	19413114	1048187
Hits	5442607	85242126	18155152	1051148	19412833	1041728
Stream Hits	1343	167979	772	256	3	226
Misses	33960	2632754	247331	5436	278	6233
Copybacks	23927	1131248	99779	3877	64	4135
Ratio	99.38	97.01	98.66	99.49	100	99.41
Cycles used	1745528	98938498	7830840	804973	6759	748168
Conflicts	13233	1080678	24933	1702	109	1759
Bus Utilization	0.107616	0.310699	0.138699	0.068223	0.00007	0.063517
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0

Table 9: Experimental Data Two Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
exit	1	1	1	1	1	1
read	33	207	128	0	0	0
write	8027	33567	27880	539	0	523
open	0	28	24	0	0	0
close	0	15	11	0	0	0
brk	4	10	2	2	2	2
time	0	1	1	0	0	0
times	0	0	0	0	0	0
sysconf	0	0	0	0	0	0
ioctl	3	17	15	0	0	0
lseek	0	0	1	0	0	0
fstat	2	16	14	0	0	0
getpid	0	1	1	0	0	0
fcntl	0	0	0	4	4	4
access	0	0	1	0	0	0
creat	0	0	0	0	0	0
unlink	0	0	1	0	0	0
stat	0	0	0	0	0	0
lstat	0	0	0	0	0	0
Updates	541449	10958780	1628113	73617	1300713	78905
Bytes Allocated	24370304	457537632	64888560	5724832	48020800	5849392
Lines Allocated	869433	15874725	2282417	187998	1750761	193435
Bytes Deallocated	24370256	457537056	64888512	5724784	48020752	5849344
Lines Deallocated	869431	15874706	2282415	187996	1750759	193433
Max Depth	2000	6672	2944	6928	544	6960

Table 9: Experimental Data Two Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Max Lines	65	212	95	220	19	222
Mallocs	8353	111509	2897	271	25	271
Bytes Malloc	101357	3394867	324299	54278	3018	36482
Lines Malloc	9493	171842	11230	1790	104	1257
Frees	8481	94622	2931	259	21	259
Jsr/Bsr	341056	8841692	1193513	84818	1578118	88562
Jmp r1	325149	7531295	1054927	57054	1328085	59394
Leaf Nodes	229405	4593273	710450	43285	1027836	43540

Table 9: Experimental Data Two Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.2.3 Experiment Two Part Three Data

Parameter	hsim	mpsdemo	z80sim	photon
Cycles	50282309	1505153	167140237	78878275
Instructions	51083648	1035032	168680934	66728571
IPC	1.015937	0.687659	1.009218	0.845969
Two Inst	15421157	250424	49402361	19038341
One Inst	20241334	534184	69876212	28651889
Zero Inst	14619818	720545	47861664	31188045
No Inst	8306349	620587	42338677	22544362
Source Unav	17449962	338583	38435865	19816338
Destination Unav	202789	28073	226297	1588499
Pipe Full	0	0	0	0
Reservation Full	202285	9386	1580750	1807562
DMU B/W	6729246	188616	29597510	9453608
Empty D Slot	1261569	59377	5084051	2793474
Branch-Branch	706002	8110	394744	970564
History Full	0	68	0	10
Serialized	971	913	19	409895
Carry bit	0	0	0	0
Decode Error	0	0	0	0
	1	1	1	1
Integer 1	18035714	350624	60896769	24970964
Integer 1 %	35.3	33.9	36.1	37.4
	2	2	2	2
Integer 2	1840245	31267	7040142	2582983
Integer 2 %	3.6	3	4.2	3.9
Load	14533102	328880	45227784	14940641
Load %	28.4	31.8	26.8	22.4
Store	4094789	151389	25616181	9598204

Table 10: Experimental Data Two Part Three

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Store %	8	14.6	15.2	14.4
Float Add	0	0	0	819012
Float Add %	0	0	0	1.2
Float Mul	953352	476	0	933456
Float Mul %	1.9	0	0	1.4
Float Div	201380	283	0	355089
Float Div %	0.4	0	0	0.5
Graph Add	0	0	0	0
Graph Add %	0	0	0	0
Graph Bit	0	0	0	0
Graph Bit %	0	0	0	0
Cond Branch	8593194	91109	11016342	7661540
Cond Branch %	16.8	8.8	6.5	11.5
Uncond Branch	2830918	80536	18883707	4743408
Uncond Branch %	5.5	7.8	11.2	7.1
Trap	952	466	7	36872
Trap %	0	0	0	0.1
Rte	0	0	0	0
Rte %	0	0	0	0
Control	2	2	2	86402
Control %	0	0	0	0.1
Load Latency	2.2	2.65	2.35	2.56
Load Clocks	1.91	2.07	1.81	1.96
Load hits	0	1	0	0
Decoupled Loads	76	1568	24648	104448
Decoupled Stores	12	89	4	1757
address alias	51	12029	202896	456459
ld input full	21	77	0	26065

Table 10: Experimental Data Two Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
ld output full	0	0	0	0
st reservation full	202264	9309	1580750	1781497
Cond store	86	555	0	17786
Inst/Branch	4.47	6.03	5.64	5.38
BTC hits	4482823	21239	1945204	1734374
BTC misses	2015	57947	1196668	3112904
Correct Pred	4430702	46752	5672435	3396769
Incorrect Pred	1694922	26124	400522	1450616
History Depth	1.25	0.91	0.46	0.96
Flush History	1313908	16472	185385	1059487
Flush Stall	0	0	0	0
Accesses	35504442	704515	122725551	47838763
Hits	35499273	639571	120328223	45820666
Stream Hits	1962	12362	491494	509965
Misses	3207	52582	1905834	1508132
Ratio	99.99	92.54	98.45	96.85
Accesses	17719803	472401	70825524	24099198
Hits	17719564	467007	70806364	23942889
Stream Hits	1	111	5	1670
Misses	238	5283	19155	154639
Copybacks	25	2753	6573	66967
Ratio	100	98.88	99.97	99.36
Cycles used	28506	570127	17795624	15460607
Conflicts	61	2928	9907	36601
Bus Utilization	0.000567	0.378783	0.106471	0.196006
Accesses	0	0	0	0
Misses	0	0	0	0
Ratio	0	0	0	0

Table 10: Experimental Data Two Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Accesses	0	0	0	0
Misses	0	0	0	0
Ratio	0	0	0	0
exit	1	1	1	1
read	936	14	0	8
write	7	433	0	849
open	1	6	0	4
close	0	6	0	4
brk	2	2	2	2
time	0	0	0	0
times	0	0	0	0
sysconf	0	0	0	0
ioctl	0	0	0	0
lseek	0	0	0	0
fstat	0	0	0	0
getpid	0	0	0	0
fcntl	5	4	4	4
access	0	0	0	0
creat	0	0	0	0
unlink	0	0	0	0
stat	0	0	0	0
lstat	0	0	0	0
Updates	1616353	75240	9008479	2966768
Bytes Allocated	51728576	1613440	548799536	130314976
Lines Allocated	1718440	58819	18022579	4313815
Bytes Deallocated	51728528	1613392	548799488	130314928
Lines Deallocated	1718438	58817	18022577	4313813

Table 10: Experimental Data Two Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Max Depth	560	1920	784	32448
Max Lines	19	65	27	1017
Mallocs	13	172	10	3730
Bytes Malloc	5280	27560	68221	799360
Lines Malloc	167	929	2133	25160
Frees	0	139	0	4
Jsr/Bsr	1112760	33345	6833472	1713952
Jmp r1	1011636	27570	6833468	1453952
Leaf Nodes	505713	15235	4805653	1112475

Table 10: Experimental Data Two Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.3 Experiment Three

C.3.1 Experiment Three Part One Data

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Cycles	19510051	19726812	587493	6211244	26003858	44639517
Instructions	23562853	23672337	527195	6886673	23504749	47417814
IPC	1.207729	1.200008	0.897364	1.108743	0.903895	1.062239
Two Inst	6456723	6492832	158059	2095328	7376352	15275116
One Inst	10649407	10686673	211077	2696017	8752045	16867582
Zero Inst	2403921	2547307	218357	1419899	9875461	12496819
No Inst	2372283	4551023	202886	1219084	8036491	11650344
Source Unav	8462933	8496547	125165	1810112	6978833	11051950
Destination Unav	11011	23322	3168	1157	179633	71400
Pipe Full	0	0	0	0	0	0
Reservation Full	4160	27964	2411	10496	162191	911437
DMU B/W	2148717	73236	60943	453729	1929737	3371785
Empty D Slot	33485	34574	25130	353614	664384	985499
Branch-Branch	13794	17824	7930	262490	509430	1255903
History Full	70	52	12	125	8478	617
Serialized	5103	5397	206	4863	7429	60933
Carry bit	0	0	0	0	49	3
Decode Error	0	0	0	0	0	0
	1	1	1	1	1	1
Integer 1	10660261	10712464	193319	2465411	9224933	18740482
Integer 1 %	45.2	45.3	36.7	35.8	39.2	39.5
	2	2	2	2	2	2
Integer 2	2105469	2104720	20254	192424	990546	1668474
Integer 2 %	8.9	8.9	3.8	2.8	4.2	3.5
Load	6375910	6386883	125774	1439755	4969920	9542619

Table 11: Experimental Data Three Part One

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Load %	27.1	27	23.9	20.9	21.1	20.1
Store	2159954	2186798	64034	904271	2699595	5629296
Store %	9.2	9.2	12.1	13.1	11.5	11.9
Float Add	0	0	0	0	32825	0
Float Add %	0	0	0	0	0.1	0
Float Mul	4622	2776	2475	0	59956	2758
Float Mul %	0	0	0.5	0	0.3	0
Float Div	162	791	0	12	34147	18237
Float Div %	0	0	0	0	0.1	0
Graph Add	0	0	0	0	0	0
Graph Add %	0	0	0	0	0	0
Graph Bit	0	0	0	0	0	0
Graph Bit %	0	0	0	0	0	0
Cond Branch	2196496	2200886	92430	1249258	3538070	8185907
Cond Branch %	9.3	9.3	17.5	18.1	15.1	17.3
Uncond Branch	57314	74350	28806	633100	1951024	3599586
Uncond Branch %	0.2	0.3	5.5	9.2	8.3	7.6
Trap	2663	2667	101	2440	3731	30453
Trap %	0	0	0	0	0	0.1
Rte	0	0	0	0	0	0
Rte %	0	0	0	0	0	0
Control	2	2	2	2	2	2
Control %	0	0	0	0	0	0
Load Latency	2.67	2.69	2.47	2.43	3.23	2.6
Load Clocks	2	2.01	1.95	1.95	2.33	2.04
Load hits	0	0	0	0	0	2
Decoupled Loads	1778	4120	838	425	41654	16207
Decoupled Stores	61	187	11	1086	13432	9756

Table 11: Experimental Data Three Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
address alias	124	175	66	228	19170	21444
ld input full	521	10552	141	259	5597	3382
ld output full	0	0	0	0	0	0
st reservation full	3639	17412	2270	10237	156594	908055
Cond store	59	94	70	60	10632	1746
Inst/Branch	10.45	10.4	4.35	3.66	4.28	4.02
BTC hits	28647	22790	20566	356666	804374	2301598
BTC misses	13339	20575	10064	91637	434652	679193
Correct Pred	46475	42824	33588	705425	1474576	4108299
Incorrect Pred	24911	24047	15713	180893	755948	1142035
History Depth	0.63	0.61	0.89	1.44	0.9	1.02
Flush History	13002	11707	11249	176515	532155	856043
Flush Stall	0	0	0	0	0	0
Accesses	17083046	17152035	348928	4629901	16131631	32218591
Hits	17055967	17111772	330556	4623268	15538935	31640803
Stream Hits	6685	10041	3892	1821	151053	168751
Misses	20394	30222	14480	4812	441643	409037
Ratio	99.88	99.82	95.85	99.9	97.26	98.73
Accesses	8532834	8571398	186498	2257437	7414627	14836211
Hits	8529378	8559532	185395	2254729	7218847	14763888
Stream Hits	31	67	15	23	4061	1375
Misses	3425	11799	1088	2685	191719	70948
Copybacks	1846	5131	567	1933	87305	31000
Ratio	99.96	99.86	99.42	99.88	97.41	99.52
Cycles used	247678	421641	156386	79931	6376233	4793495
Conflicts	1996	2708	607	338	62646	21697
Bus Utilization	0.012695	0.021374	0.266192	0.012869	0.245203	0.107382
Accesses	0	0	0	0	0	0

Table 11: Experimental Data Three Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
exit	1	1	1	1	1	1
read	2154	2150	7	30	23	55
write	0	6	85	2381	3692	30379
open	84	84	1	5	0	2
close	84	85	1	5	0	1
brk	2	2	2	4	10	8
time	0	0	0	0	0	0
times	0	0	0	0	0	0
sysconf	0	0	0	0	0	0
ioctl	83	83	0	7	3	4
lseek	0	0	0	0	0	0
fstat	83	83	0	7	2	3
getpid	0	0	0	0	0	0
fcntl	4	4	4	0	0	0
access	0	0	0	0	0	0
creat	0	1	0	0	0	0
unlink	0	0	0	0	0	0
stat	0	0	0	0	0	0
lstat	168	168	0	0	0	0
Updates	25919	37625	13959	187111	808767	1473699
Bytes Allocated	1179408	1412352	543520	7432752	29473392	77343984
Lines Allocated	39902	49857	18704	276215	1027620	2593692
Bytes Deallocated	1179360	1412304	543472	7432704	29473344	77343936

Table 11: Experimental Data Three Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Lines Deallocated	39900	49855	18702	276213	1027618	2593690
Max Depth	3760	3936	2224	2192	2864	2176
Max Lines	121	125	73	71	95	71
Mallocs	439	603	187	396	4861	6844
Bytes Malloc	355494	378215	9859	53962	285455	206038
Lines Malloc	11312	12025	382	1933	11452	10324
Frees	255	589	2	331	4867	6836
Jsr/Bsr	28592	36654	15524	98335	712172	1443010
Jmp r1	22547	31814	9273	96955	677585	1215922
Leaf Nodes	12365	18241	4570	89482	433657	968084

Table 11: Experimental Data Three Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.3.2 Experiment Three Part Two Data

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Cycles	16934597	371279568	59337227	12012310	99535273	12036624
Instructions	17934096	286013430	58546384	6510246	73327244	6523335
IPC	1.059021	0.770345	0.986672	0.541965	0.736696	0.541957
Two Inst	5656007	87522023	18098333	1981555	21315413	1925318
One Inst	6622082	110969384	22349718	2547136	30696418	2672699
Zero Inst	4656508	172788161	18889176	7483619	47523442	7438607
No Inst	4031257	141246620	14977396	1996663	11068654	2077384
Source Unav	4158614	94317117	18323352	7297349	45287927	7300337
Destination Unav	14613	2547197	106665	37923	6402785	35854
Pipe Full	0	0	0	0	0	0
Reservation Full	198249	2230467	385626	87898	250167	78360
DMU B/W	1566178	25882716	4720096	356137	6842853	351939
Empty D Slot	674538	9437117	1257410	179895	849175	183698
Branch-Branch	607850	7626318	1385082	35746	630491	33359
History Full	2590	31807	374	1065	0	436
Serialized	16126	67850	56159	15412	3050184	18709
Carry bit	2183	4854	1	0	0	0
Decode Error	0	0	0	0	0	0
	1	1	1	1	1	1
Integer 1	6582558	108411579	22546145	2668756	21062381	2697461
Integer 1 %	36.7	37.9	38.5	41	28.7	41.4
	2	2	2	2	2	2
Integer 2	685078	8999181	1770170	190296	3825351	169245
Integer 2 %	3.8	3.1	3	2.9	5.2	2.6
Load	3805567	64684717	13138664	726251	14156983	718681
Load %	21.2	22.6	22.4	11.2	19.3	11
Store	2396924	33721072	6608924	359972	6552928	360857

Table 12: Experimental Data Three Part Two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Store %	13.4	11.8	11.3	5.5	8.9	5.5
Float Add	0	95739	4	207843	11776317	214555
Float Add %	0	0	0	3.2	16.1	3.3
Float Mul	4706	486669	19355	797540	5317973	796161
Float Mul %	0	0.2	0	12.3	7.3	12.2
Float Div	1246	757522	139028	274602	570988	274317
Float Div %	0	0.3	0.2	4.2	0.8	4.2
Graph Add	0	0	0	0	0	0
Graph Add %	0	0	0	0	0	0
Graph Bit	0	0	0	0	0	0
Graph Bit %	0	0	0	0	0	0
Cond Branch	3061765	37957241	8991478	1019892	5314313	1018952
Cond Branch %	17.1	13.3	15.4	15.7	7.2	15.6
Uncond Branch	1388177	30865674	5304514	260551	3899950	267610
Uncond Branch %	7.7	10.8	9.1	4	5.3	4.1
Trap	8073	34034	28100	1721	250022	1990
Trap %	0	0	0	0	0.3	0
Rte	0	0	0	0	0	0
Rte %	0	0	0	0	0	0
Control	2	2	2	2822	600038	3506
Control %	0	0	0	0	0.8	0.1
Load Latency	2.64	3.24	2.46	2.35	2.69	2.38
Load Clocks	1.99	2.3	1.99	1.95	2.26	1.96
Load hits	1	67	0	0	0	1
Decoupled Loads	3334	426453	10297	3055	110	2207
Decoupled Stores	2419	137922	9112	53	3	59
address alias	11883	246815	30537	8353	3449496	9584
ld input full	1036	175940	15332	108	50002	209

Table 12: Experimental Data Three Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Id output full	0	0	0	0	0	0
st reservation full	197213	2054527	370294	87790	200165	78151
Cond store	1422	39589	789	455	62	746
Inst/Branch	4.03	4.16	4.1	5.08	7.96	5.07
BTC hits	1154037	9828905	2378548	234675	1626627	200116
BTC misses	251207	7955832	866809	129679	100228	129685
Correct Pred	1504392	14267319	3855549	424367	2579608	444239
Incorrect Pred	391861	8133772	1649333	229803	1848466	232682
History Depth	1.52	0.62	0.78	0.99	1.98	1.02
Flush History	404077	3812685	921689	136745	2435721	133669
Flush Stall	0	0	0	0	0	0
Accesses	12584755	196220924	40746582	4490807	52690539	4512727
Hits	12382024	184011279	40099102	4347190	52489872	4379883
Stream Hits	54196	2994619	139468	39395	178	33341
Misses	148535	9215026	508012	104222	200489	99503
Ratio	98.82	95.3	98.75	97.68	99.62	97.8
Accesses	5854017	96036308	19365296	1063614	19654884	1056399
Hits	5834929	93159394	19302662	1057633	19654635	1050751
Stream Hits	824	148392	246	244	1	392
Misses	18264	2728522	62388	5737	248	5256
Copybacks	10469	1063986	28049	4114	27	3642
Ratio	99.69	97.16	99.68	99.46	100	99.5
Cycles used	1631608	123379888	5398451	1038760	1606729	1007702
Conflicts	3747	1134614	12676	1997	83	1760
Bus Utilization	0.096348	0.33231	0.090979	0.086475	0.016142	0.08372
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0

Table 12: Experimental Data Three Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
exit	1	1	1	1	1	1
read	33	207	128	0	0	0
write	8027	33567	27880	539	0	523
open	0	28	24	0	0	0
close	0	15	11	0	0	0
brk	7	181	22	2	2	2
time	0	1	1	0	0	0
times	0	0	0	0	0	0
sysconf	0	0	0	0	0	0
ioctl	3	17	15	0	0	0
lseek	0	0	1	0	0	0
fstat	2	16	14	0	0	0
getpid	0	1	1	0	0	0
fcntl	0	0	0	4	4	4
access	0	0	1	0	0	0
creat	0	0	0	0	0	0
unlink	0	0	1	0	0	0
stat	0	0	0	0	0	0
lstat	0	0	0	0	0	0
Updates	697345	15653680	2468995	79283	1755729	82837
Bytes Allocated	28784080	571564640	84828160	5860112	56542032	5951616
Lines Allocated	1032770	20179644	3103342	193682	2155784	197321
Bytes Deallocated	28784032	571564032	84828112	5860064	56541984	5951568
Lines Deallocated	1032768	20179624	3103340	193680	2155782	197319
Max Depth	2096	6752	2944	6928	496	6960

Table 12: Experimental Data Three Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Max Lines	67	214	95	220	18	222
Mallocs	8358	111204	2912	272	25	272
Bytes Malloc	134125	4179147	398027	56878	3018	39082
Lines Malloc	10517	195988	13534	1872	104	1339
Frees	8491	94622	2939	259	21	259
Jsr/Bsr	437547	13078083	1996390	92979	1805614	97333
Jmp rl	408316	12553982	1896575	66367	1555577	69363
Leaf Nodes	260465	7884215	1447282	48832	1027860	49230

Table 12: Experimental Data Three Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.3.3 Experiment Three Part Three Data

Parameter	hsim	mpsdemo	z80sim	photon
Cycles	55806372	2011710	381870711	83191087
Instructions	54903727	1311059	386666663	68854489
IPC	0.983825	0.651714	1.012559	0.827667
Two Inst	17036058	326079	122308748	19550530
One Inst	20831611	658901	142049167	29753429
Zero Inst	17938703	1026730	117512796	33887128
No Inst	11645308	961439	128065732	25338128
Source Unav	18327059	339936	73244445	20315227
Destination Unav	203166	28731	1204619	1452117
Pipe Full	0	0	0	0
Reservation Full	202639	9923	43111	1861387
DMU B/W	6521239	240069	51512927	9928346
Empty D Slot	1161111	93746	5405449	2970476
Branch-Branch	705934	9746	31034	959020
History Full	2	91	9	286
Serialized	1876	921	14	404397
Carry bit	0	0	0	0
Decode Error	0	0	0	0
	1	1	1	1
Integer 1	19647960	459071	149193017	25733598
Integer 1 %	35.8	35	38.6	37.4
	2	2	2	2
Integer 2	1840305	20712	23754359	2733798
Integer 2 %	3.4	1.6	6.1	4
Load	15133026	360982	81327482	15338745
Load %	27.6	27.5	21	22.3
Store	4697732	225279	55589795	10047566

Table 13: Experimental Data Three Part Three

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Store %	8.6	17.2	14.4	14.6
Float Add	0	0	0	819012
Float Add %	0	0	0	1.2
Float Mul	953352	481	0	933456
Float Mul %	1.7	0	0	1.4
Float Div	201380	283	0	355089
Float Div %	0.4	0	0	0.5
Graph Add	0	0	0	0
Graph Add %	0	0	0	0
Graph Bit	0	0	0	0
Graph Bit %	0	0	0	0
Cond Branch	8794181	92372	15545027	7667911
Cond Branch %	16	7	4	11.1
Uncond Branch	3634837	151411	61256974	5102040
Uncond Branch %	6.6	11.5	15.8	7.4
Trap	952	466	7	36872
Trap %	0	0	0	0.1
Rte	0	0	0	0
Rte %	0	0	0	0
Control	2	2	2	86402
Control %	0	0	0	0.1
Load Latency	2.2	2.69	2.29	2.56
Load Clocks	1.9	2.09	1.81	1.96
Load hits	0	0	0	0
Decoupled Loads	110	1683	58	104443
Decoupled Stores	14	68	6	2014
address alias	48	18764	31	449060
ld input full	129	216	4	27096

Table 13: Experimental Data Three Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
ld output full	0	0	0	0
st reservation full	202510	9707	43107	1834291
Cond store	45	231	85	19762
Inst/Branch	4.42	5.38	5.03	5.39
BTC hits	4695030	28832	11761060	1787460
BTC misses	102294	94775	1726388	3290628
Correct Pred	4538684	46501	5630591	3393402
Incorrect Pred	1787905	25007	178767	1458322
History Depth	1.07	0.85	0	0.91
Flush History	1210393	14486	110	1030677
Flush Stall	0	0	0	0
Accesses	37392961	871047	266854939	48802673
Hits	36992161	773533	258333350	46415435
Stream Hits	148	15164	1835900	540112
Misses	400652	82350	6685689	1847126
Ratio	98.93	90.55	97.49	96.22
Accesses	18925684	579736	136917213	24990683
Hits	18925343	573916	136863222	24831128
Stream Hits	4	204	3	2022
Misses	337	5616	53988	157533
Copybacks	96	3006	12723	67269
Ratio	100	99.03	99.96	99.37
Cycles used	3242431	886858	65723579	18411787
Conflicts	65	3634	39954	37695
Bus Utilization	0.058101	0.440848	0.17211	0.221319
Accesses	0	0	0	0
Misses	0	0	0	0
Ratio	0	0	0	0

Table 13: Experimental Data Three Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Accesses	0	0	0	0
Misses	0	0	0	0
Ratio	0	0	0	0
exit	1	1	1	1
read	936	14	0	8
write	7	433	0	849
open	1	6	0	4
close	0	6	0	4
brk	2	2	2	2
time	0	0	0	0
times	0	0	0	0
sysconf	0	0	0	0
ioctl	0	0	0	0
lseek	0	0	0	0
fstat	0	0	0	0
getpid	0	0	0	0
fcntl	5	4	4	4
access	0	0	0	0
creat	0	0	0	0
unlink	0	0	0	0
stat	0	0	0	0
lstat	0	0	0	0
Updates	2420269	149390	35995867	3396760
Bytes Allocated	72630432	2814976	1078619344	138393872
Lines Allocated	2522357	106408	39719464	4622553
Bytes Deallocated	72630384	2814928	1078619296	138393824
Lines Deallocated	2522355	106406	39719462	4622551
Max Depth	624	1760	768	32432

Table 13: Experimental Data Three Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Max Lines	21	57	29	1017
Mallocs	13	170	10	3730
Bytes Malloc	5280	27520	68221	799360
Lines Malloc	167	927	2133	25160
Frees	0	138	0	4
Jsr/Bsr	1414232	58731	29464086	1844138
Jmp r1	1413595	52856	29464064	1592241
Leaf Nodes	706691	29928	14538581	1188410

Table 13: Experimental Data Three Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.4 Experiment Four

C.4.1 Experiment Four Part One Data

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Cycles	19833537	20072188	566440	5925089	23826800	39396884
Instructions	23628967	23739901	516657	6846589	22389674	41377387
IPC	1.191364	1.182726	0.912112	1.155525	0.939684	1.050271
Two Inst	6490009	6519025	148610	2247469	6981401	13028370
One Inst	10648949	10701851	219437	2351651	8426872	15320647
Zero Inst	2694579	2851312	198393	1325969	8418527	11047867
No Inst	2352595	2475118	179794	780942	6784653	9790378
Source Unav	8596746	8654412	126185	1637772	6395534	9597650
Destination Unav	22141	21342	3730	992	222698	68910
Pipe Full	0	0	0	0	0	0
Reservation Full	160155	169647	3941	15669	175118	927953
DMU B/W	2148703	2149175	66520	621275	2002917	2665419
Empty D Slot	35259	46581	25487	349744	608929	1326505
Branch-Branch	16636	18301	9200	264261	504759	1906218
History Full	3633	7756	1035	1884	43869	20782
Serialized	5687	6377	203	4874	7451	60908
Carry bit	0	0	0	0	305	5
Decode Error	0	0	0	0	0	0
	1	1	1	1	1	1
Integer 1	10696485	10748544	194291	2367274	8771892	15309707
Integer 1 %	45.3	45.3	37.6	34.6	39.2	37
	2	2	2	2	2	2
Integer 2	2105733	2105267	14895	274060	902518	1153707
Integer 2 %	8.9	8.9	2.9	4	4	2.8

Table 14: Experimental Data Four Part One

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Load	6372919	6382472	124599	1515116	4834782	8888203
Load %	27	26.9	24.1	22.1	21.6	21.5
Store	2178575	2206608	62849	896888	2583067	4974610
Store %	9.2	9.3	12.2	13.1	11.5	12
Float Add	0	0	0	0	32849	0
Float Add %	0	0	0	0	0.1	0
Float Mul	4622	2776	2475	0	59956	2758
Float Mul %	0	0	0.5	0	0.3	0
Float Div	162	791	0	12	34147	18237
Float Div %	0	0	0	0	0.2	0
Graph Add	0	0	0	0	0	0
Graph Add %	0	0	0	0	0	0
Graph Bit	0	0	0	0	0	0
Graph Bit %	0	0	0	0	0	0
Cond Branch	2218637	2225732	92514	1249183	3509236	8129444
Cond Branch %	9.4	9.4	17.9	18.2	15.7	19.6
Uncond Branch	49169	65042	24931	541614	1657502	2870272
Uncond Branch %	0.2	0.3	4.8	7.9	7.4	6.9
Trap	2663	2667	101	2440	3723	30447
Trap %	0	0	0	0	0	0.1
Rte	0	0	0	0	0	0
Rte %	0	0	0	0	0	0
Control	2	2	2	2	2	2
Control %	0	0	0	0	0	0
Load Latency	2.72	2.74	2.56	2.68	3.04	2.54
Load Clocks	2.03	2.03	1.98	2.01	2.27	2.04
Load hits	0	0	0	0	0	1
Decoupled Loads	3961	4313	827	1790	54002	34412

Table 14: Experimental Data Four Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Decoupled Stores	2318	2340	21	1008	4980	8377
address alias	843	803	145	257	14709	31249
ld input full	3272	2636	252	197	13095	3024
ld output full	0	0	0	0	0	0
st reservation full	156883	167011	3689	15472	162023	924929
Cond store	90	165	87	0	13534	4004
Inst/Branch	10.42	10.36	4.4	3.82	4.33	3.76
BTC hits	39806	45462	16809	522739	939980	2880697
BTC misses	11064	21007	8934	4183	328545	532650
Correct Pred	68898	55293	37301	621136	1497323	3935008
Incorrect Pred	25594	24329	15918	96860	746925	1282914
History Depth	0.67	0.63	0.9	2.67	1.26	1.44
Flush History	14053	11840	11502	173813	683268	1122425
Flush Stall	0	0	0	0	0	0
Accesses	15064996	15116200	348828	4954882	15594515	27741452
Hits	15042623	15078258	333469	4951189	15135625	27288856
Stream Hits	5333	9389	3266	820	129389	113072
Misses	17040	28553	12093	2873	329501	339524
Ratio	99.89	99.81	96.53	99.94	97.89	98.78
Accesses	8547636	8586113	183636	2241931	7105643	13505844
Hits	8511485	8542526	181964	2238157	6938712	13414912
Stream Hits	142	344	12	13	1713	1970
Misses	36009	43243	1660	3761	165218	88962
Copybacks	20976	25332	930	2901	62654	39908
Ratio	99.58	99.5	99.1	99.83	97.67	99.34
Cycles used	602081	796062	133554	77395	4871091	4022818
Conflicts	3567	5030	902	378	47143	27874
Bus Utilization	0.030357	0.03966	0.235778	0.013062	0.204437	0.10211

Table 14: Experimental Data Four Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
exit	1	1	1	1	1	1
read	2154	2150	7	30	23	55
write	0	6	85	2381	3692	30379
open	84	84	1	5	0	2
close	84	85	1	5	0	1
brk	2	2	2	4	2	2
time	0	0	0	0	0	0
times	0	0	0	0	0	0
sysconf	0	0	0	0	0	0
ioctl	83	83	0	7	3	4
lseek	0	0	0	0	0	0
fstat	83	83	0	7	2	3
getpid	0	0	0	0	0	0
fcntl	4	4	4	0	0	0
access	0	0	0	0	0	0
creat	0	1	0	0	0	0
unlink	0	0	0	0	0	0
stat	0	0	0	0	0	0
lstat	168	168	0	0	0	0
Updates	19393	29593	11629	181889	733171	837945
Bytes Allocated	1018656	1216720	483520	7260032	26948016	57520704
Lines Allocated	33377	41826	16376	270579	948050	1953723

Table 14: Experimental Data Four Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	chkbom	genbom	listbom	soelim	pic	eqn
Bytes Deallocated	1018608	1216672	483472	7259984	26947968	57520656
Lines Deallocated	33375	41824	16374	270577	948048	1953721
Max Depth	3776	3952	2224	2112	2912	2160
Max Lines	121	126	73	68	96	70
Mallocs	439	603	187	393	4857	6834
Bytes Malloc	355494	378215	9859	33482	251743	150742
Lines Malloc	11312	12025	382	1293	10400	8596
Frees	255	589	2	331	4864	6841
Jsr/Bsr	24735	31989	13851	95601	597361	1081712
Jmp r1	18532	27508	7586	94575	529167	644857
Leaf Nodes	11105	17585	3469	87786	329458	449945

Table 14: Experimental Data Four Part One (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.4.2 Experiment Four Part Two Data

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Cycles	16300602	320882214	57250369	11823795	97991711	11811652
Instructions	16980002	245698439	52716443	6479078	72046618	6483479
IPC	1.041679	0.765697	0.920805	0.547969	0.735232	0.548905
Two Inst	5319228	73446124	16225976	1961999	21455468	1913516
One Inst	6341546	98806191	20264491	2555080	29135682	2656447
Zero Inst	4639828	148629899	20759902	7306716	47400561	7241689
No Inst	4226259	112224162	14064970	1768335	10099958	1820279
Source Unav	4049264	83065842	18577759	7307594	44627508	7320094
Destination Unav	23098	4717865	378274	38112	6427824	42803
Pipe Full	0	0	0	0	0	0
Reservation Full	193226	4440329	604347	104632	350709	94323
DMU B/W	1135396	26834105	4526532	358402	6474028	358599
Empty D Slot	652533	7382595	1237035	183875	899179	183174
Branch-Branch	634951	7938780	1429734	54357	655431	30993
History Full	43429	376297	11813	3493	94	2115
Serialized	16145	67694	56174	15585	3000193	17971
Carry bit	2242	4483	1	0	0	0
Decode Error	0	0	0	0	0	0
	1	1	1	1	1	1
Integer 1	6188956	87615035	19621693	2651052	20513166	2673609
Integer 1 %	36.4	35.7	37.2	40.9	28.5	41.2
	2	2	2	2	2	2
Integer 2	676757	7226791	1398446	195099	3878704	176938
Integer 2 %	4	2.9	2.7	3	5.4	2.7
Load	3472786	60583861	12711461	721943	14079387	715664
Load %	20.5	24.7	24.1	11.1	19.5	11
Store	2222391	30442157	6143775	362790	6275591	358545

Table 15: Experimental Data Four Part Two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Store %	13.1	12.4	11.7	5.6	8.7	5.5
Float Add	0	95739	4	207708	11801319	214414
Float Add %	0	0	0	3.2	16.4	3.3
Float Mul	4706	486536	19355	796831	5317973	795374
Float Mul %	0	0.2	0	12.3	7.4	12.3
Float Div	1246	757367	139028	274603	570988	274317
Float Div %	0	0.3	0.3	4.2	0.8	4.2
Graph Add	0	0	0	0	0	0
Graph Add %	0	0	0	0	0	0
Graph Bit	0	0	0	0	0	0
Graph Bit %	0	0	0	0	0	0
Cond Branch	3047969	37192991	8996855	1022380	5314469	1020366
Cond Branch %	18	15.1	17.1	15.8	7.4	15.7
Uncond Branch	1357119	21264097	3657744	242129	3444961	248756
Uncond Branch %	8	8.7	6.9	3.7	4.8	3.8
Trap	8070	33863	28080	1721	250022	1990
Trap %	0	0	0.1	0	0.3	0
Rte	0	0	0	0	0	0
Rte %	0	0	0	0	0	0
Control	2	2	2	2822	600038	3506
Control %	0	0	0	0	0.8	0.1
Load Latency	2.52	3.19	2.88	2.4	2.7	2.43
Load Clocks	1.98	2.27	2.16	1.95	2.28	1.99
Load hits	1	94	1	0	0	0
Decoupled Loads	31281	651949	24269	2704	79	3849
Decoupled Stores	1706	123574	10493	55	5	45
address alias	12751	89862	27872	8558	3699484	9795
ld input full	3131	349520	11488	127	50009	52

Table 15: Experimental Data Four Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
ld output full	0	0	0	0	0	0
st reservation full	190095	4090809	592859	104505	300700	94271
Cond store	2974	42461	2291	267	52	583
Inst/Branch	3.85	4.2	4.17	5.12	8.23	5.11
BTC hits	771417	8394910	1976077	234755	1293584	213494
BTC misses	364594	5825941	843495	129430	327863	107871
Correct Pred	1509043	14675540	4311025	422653	2526440	442953
Incorrect Pred	522729	9113209	1833882	229889	1973406	236336
History Depth	1.13	0.68	0.94	1.04	1.97	1.04
Flush History	404478	4953764	1176633	140526	2610700	142259
Flush Stall	0	0	0	0	0	0
Accesses	11592801	171809779	36719094	4580568	50362771	4493260
Hits	11393066	162234148	35962647	4476029	50112212	4391222
Stream Hits	64607	2436080	182002	28282	50142	30011
Misses	135128	7139551	574445	76257	200417	72027
Ratio	98.83	95.84	98.44	98.34	99.6	98.4
Accesses	5505942	88430287	18427536	1060697	19413334	1050843
Hits	5467890	85608060	18136936	1053522	19413056	1042762
Stream Hits	1454	168773	612	293	4	204
Misses	36598	2653454	289988	6882	274	7877
Copybacks	25763	1159115	154530	5069	54	5535
Ratio	99.34	97	98.43	99.35	100	99.25
Cycles used	1694327	100520399	8640023	822430	1606529	778266
Conflicts	12214	1118318	36343	1797	100	1995
Bus Utilization	0.103943	0.313263	0.150916	0.069557	0.016395	0.06589
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0

Table 15: Experimental Data Four Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Accesses	0	0	0	0	0	0
Misses	0	0	0	0	0	0
Ratio	0	0	0	0	0	0
exit	1	1	1	1	1	1
read	33	207	128	0	0	0
write	8027	33567	27880	539	0	523
open	0	28	24	0	0	0
close	0	15	11	0	0	0
brk	4	10	2	2	2	2
time	0	1	1	0	0	0
times	0	0	0	0	0	0
sysconf	0	0	0	0	0	0
ioctl	3	17	15	0	0	0
lseek	0	0	1	0	0	0
fstat	2	16	14	0	0	0
getpid	0	1	1	0	0	0
fcntl	0	0	0	4	4	4
access	0	0	1	0	0	0
creat	0	0	0	0	0	0
unlink	0	0	1	0	0	0
stat	0	0	0	0	0	0
lstat	0	0	0	0	0	0
Updates	542375	10960548	1630769	73405	1300715	78931
Bytes Allocated	24411856	457603504	64980272	5719408	48020896	5850608
Lines Allocated	870811	15876461	2285071	187786	1750764	193473
Bytes Deallocated	24411808	457602928	64980224	5719360	48020848	5850560
Lines Deallocated	870809	15876442	2285069	187784	1750762	193471
Max Depth	2000	6672	2944	6928	544	6960

Table 15: Experimental Data Four Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	tbl	troff	grops	MatrixD	Random	MatrixF
Max Lines	65	212	95	220	19	222
Mallocs	8353	112335	2897	271	25	271
Bytes Malloc	101357	3404779	324299	54278	3018	36482
Lines Malloc	9493	172668	11230	1790	104	1257
Frees	8492	94622	2931	259	21	259
Jsr/Bsr	341207	8844652	1194219	84127	1578118	88734
Jmp r1	324926	7530330	1054853	57054	1328085	59394
Leaf Nodes	229412	4593390	710464	43291	1027836	43545

Table 15: Experimental Data Four Part Two (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

C.4.3 Experiment Four Part Three Data

Parameter	hsim	mpsdemo	z80sim	photon
Cycles	50283348	1520725	167066783	79217222
Instructions	51084303	1041816	168689520	66831183
IPC	1.015929	0.685078	1.009713	0.843645
Two Inst	15421329	253524	49404746	19042478
One Inst	20241645	534768	69880028	28746227
Zero Inst	14620374	732433	47782009	31428517
No Inst	8306655	627365	42339559	22816290
Source Unav	17449519	338494	38330256	19696072
Destination Unav	202764	26297	226358	1615221
Pipe Full	0	0	0	0
Reservation Full	203131	17151	1607163	1940029
DMU B/W	6729252	188548	29597500	9444775
Empty D Slot	1261735	59278	5086294	2828142
Branch-Branch	705961	7212	394852	971000
History Full	54	898	75	32169
Serialized	970	913	19	389484
Carry bit	0	0	0	0
Decode Error	0	0	0	0
	1	1	1	1
Integer 1	18036057	353742	60900967	25022251
Integer 1 %	35.3	34	36.1	37.4
	2	2	2	2
Integer 2	1840259	31545	7040255	2590613
Integer 2 %	3.6	3	4.2	3.9
Load	14533054	329006	45227783	14940419
Load %	28.4	31.6	26.8	22.4
Store	4094955	152745	25618314	9622106

Table 16: Experimental Data Four Part Three

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Store %	8	14.7	15.2	14.4
Float Add	0	0	0	819012
Float Add %	0	0	0	1.2
Float Mul	953352	473	0	933456
Float Mul %	1.9	0	0	1.4
Float Div	201380	283	0	355089
Float Div %	0.4	0	0	0.5
Graph Add	0	0	0	0
Graph Add %	0	0	0	0
Graph Bit	0	0	0	0
Graph Bit %	0	0	0	0
Cond Branch	8593374	92893	11018485	7690562
Cond Branch %	16.8	8.9	6.5	11.5
Uncond Branch	2830918	80661	18883707	4734401
Uncond Branch %	5.5	7.7	11.2	7.1
Trap	952	466	7	36872
Trap %	0	0	0	0.1
Rte	0	0	0	0
Rte %	0	0	0	0
Control	2	2	2	86402
Control %	0	0	0	0.1
Load Latency	2.2	2.65	2.35	2.56
Load Clocks	1.91	2.07	1.8	1.95
Load hits	0	1	0	0
Decoupled Loads	46	1445	60	56478
Decoupled Stores	4	159	9	1393
address alias	11	11239	202876	429379
ld input full	1	124	0	27157

Table 16: Experimental Data Four Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
ld output full	0	0	0	0
st reservation full	203130	17027	1607163	1912872
Cond store	92	522	0	17882
Inst/Branch	4.47	6	5.64	5.38
BTC hits	4482854	22280	1947368	1754159
BTC misses	2137	58221	1196724	3115997
Correct Pred	4430714	48060	5672446	3404260
Incorrect Pred	1694909	26372	400402	1442415
History Depth	1.25	0.91	0.46	0.96
Flush History	1313863	16593	185409	1049650
Flush Stall	0	0	0	0
Accesses	35505152	707856	122731748	47856614
Hits	35500003	642142	120334503	45819188
Stream Hits	1965	12726	491158	505323
Misses	3184	52988	1906087	1532103
Ratio	99.99	92.51	98.45	96.8
Accesses	17719963	473935	70827655	24125609
Hits	17719686	468007	70824856	23958864
Stream Hits	1	85	6	1809
Misses	276	5843	2793	164936
Copybacks	42	3351	2485	70079
Ratio	100	98.77	100	99.32
Cycles used	28767	583852	17649974	15824245
Conflicts	63	3261	149	45780
Bus Utilization	0.000572	0.38393	0.105646	0.199758
Accesses	0	0	0	0
Misses	0	0	0	0
Ratio	0	0	0	0

Table 16: Experimental Data Four Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Accesses	0	0	0	0
Misses	0	0	0	0
Ratio	0	0	0	0
exit	1	1	1	1
read	936	14	0	8
write	7	433	0	849
open	1	6	0	4
close	0	6	0	4
brk	2	2	2	2
time	0	0	0	0
times	0	0	0	0
sysconf	0	0	0	0
ioctl	0	0	0	0
lseek	0	0	0	0
fstat	0	0	0	0
getpid	0	0	0	0
fcntl	5	4	4	4
access	0	0	0	0
creat	0	0	0	0
unlink	0	0	0	0
stat	0	0	0	0
lstat	0	0	0	0
Updates	1616351	75194	9008479	2972452
Bytes Allocated	51728528	1610176	548799536	130446688
Lines Allocated	1718438	58715	18022579	4319303
Bytes Deallocated	51728480	1610128	548799488	130446640
Lines Deallocated	1718436	58713	18022577	4319301

Table 16: Experimental Data Four Part Three (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Parameter	hsim	mpsdemo	z80sim	photon
Max Depth	560	1920	784	32448
Max Lines	19	65	27	1017
Mallocs	13	172	10	3730
Bytes Malloc	5280	27572	68221	799360
Lines Malloc	167	929	2133	25160
Frees	0	137	0	4
Jsr/Bsr	1112760	33433	6833472	1704913
Jmp r1	1011636	27570	6833468	1453952
Leaf Nodes	505713	15252	4805653	1112391

Table 16: Experimental Data Four Part Three (Continued)

C.5 Experiment Five

Parameter	listbom	MatrixD	mpsdemo	z80sim
Clocks	552156	11734458	1456505	16486548 9

Table 17: Experimental Data Five

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Bibliography

[1] Todd M. Austin, Scott E. Breach and Surindar S. Sohi, "Efficient Detection of All Pointer and Array Access Errors," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[2] Tim Korson and John D. McGregor, "Understanding Object-Oriented: A Unifying Paradigm," *Communications of the ACM*, September 1990, vol. 33, no. 9, Sept. 1990.

[3] Howard D. Owens, Baxter F. Womack and Mario J. Gonzalez, "Software Error Classification Using Purify," *Proceedings of International Conference on Software Maintenance*, Monterey, California, IEEE Computer Society Press, 1996, pp 104-113.

[4] B. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, May 1973.

[5] B. Henderson-Sellers and J. Edwards, "The Object-Oriented Systems Life Cycle", *Communications of the ACM*, vol. 33, no. 9, Sept. 1990.

[6] F. Gryna, "Quality costs," *Juran's Quality Control Handbook*, 4th ed. (New York: McGraw-Hill, 1988).

[7] B. P. Lientz and E. B. Swanson, "Software maintenance: a user/management tug of war," *Data Management*, pp. 26-30, Apr. 1979.

[8] Stephen T. Knox, "Modeling the cost of software quality," *Digital Technical Journal* pp. 9-17, vol. 5, no. 4 (Fall 1993).

[9] Bertrand Meyer, *Object-oriented software construction* (Prentice Hall, 1988).

[10] J. Hager, "Software Cost Reduction Methods in Practice", *IEEE Transactions of Software Engineering*, vol. 15, no. 12, Dec. 1989.

[11] Mark Sullivan and Ram Chillarege, "Software defects and their impact on system availability - a study of field failures in operating systems," *Digest 21st International Symposium on Fault Tolerant Computing* (Montreal, June 1991).

[12] Mark Sullivan and Ram Chillarege, "A comparison of software defects in database management systems (DB2, IMS) and operating systems (MVS)," *Digest 22nd International Symposium on Fault Tolerant Computing* (Boston, July 1992).

[13] Reed Hastings and Bob Joyce, "Purify: fast detection of memory leaks and access errors," *Proceedings of the Winter Usenix Conference*, Jan 1992.

[14] *Purify user's guide*, release 2.1, Pure Software Inc. 1993.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

[15] *SunOS 5.0 multithread architecture*, a white paper, SunSoft, 1991.

[16] *Custom cell synthesizer (CCS) version 2.5 research prototype design specification: as built*, MCC technical report number: CPS-108-92(Q) (May, 1992).

[17] *Custom cell synthesizer (CCS) version 2.5 research prototype release CCS sources & sun4 binaries magnetic tape, and installation instructions*, MCC technical report number: CPS-106-92(Q) (May, 1992).

[18] *Custom cell synthesizer (CCS) version 2.5 research prototype user's guide*, MCC technical report number: CPS-107-92(Q) (May, 1992).

[19] P. Banerjee, J. Rahmeh, C. Stunkel, S. Nair, K. Roy, and J. Abraham, "An evaluation of system-level fault tolerance on the Intel hypercube multiprocessor", Coordinated Science Laboratory, University of Illinois.

[20] *InterViews reference manual*, version 3.1-beta, Stanford University 1992.

[21] John Vlissides, Steve Tang and Charles Brauer, *Ibuild user's guide*, 1992.

[22] *Solaris Application Level Multithreading Seminar: Participant's Guide*, Sun Microsystems, February 1993.

[23] Natraj Arni and KayLiang Ong, *LDL user's guide*, edition 2.0, MCC technical report number Carnot-012-93(P).

[24] *Carnot software release 2.0*, MCC technical report number Carnot-028-93(Q).

[25] Jan-Simon Pendry and Nick Williams, *The amd reference manual*, March 1991.

[26] Jose Nelson Amaral, "A parallel architecture for serializable production systems", Ph.D. qualifying exam, The University of Texas at Austin, June 1993.

[27] Anurag Acharya, "PPL: an explicitly parallel production language for large scale parallelism", *Proceedings of the IJAI-93 Workshop on Production systems and their innovative applications* (Chambery, August 93).

[28] Eric Allman, "Mail systems and addressing in 4.2bsd", *Usenix*, Jan 1983.

[29] Burton J. Smith, "A Pipelined Shared Resources MIMD Computer," *Proceedings 1978 International Conference on Parallel Processing*, 1978, pp.6-8.

[30] Kai Hwang and Faye A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.

[31] lHenry M. Levy, *Capability-Based Computer Systems*, Digital Press, 1984.

[32] Viktors Berstis, "Security And Protection Of Data In The IBM System/38," *Proceeding of the 7th Int. Symposium on Computer Architecture*, 1980, pp. 245-252.

[33] Merle E. Houdek, Frank G. Soltis, and Roy L. Homan. "IBM System/38 support for capability-based addressing," *Proceedings of the 8th Symposium on Computer*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

Architecture, pp. 341-348, 1981.

[34] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally, "Hardware Support for Fast Capability-based Addressing," *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[35] Richard S. Wiener and Lewis J. Pinson, *An Introduction to Object-Oriented Programming and C++*, Addison-Wesley, 1988.

[36] Edward F. Gehringer and Robert P. Colwell "Fast object-oriented procedure calls: Lessons from the intel 432," *The 13th Annual International Symposium on Computer Architecture*, pp. 92-101, 1986.

[37] Fred J. Pollack, George W. Cox, Dan W. Hammerstrom, Kevin C. Kahn, Konrad K. Lai, and Justin R. Rattner, "Supporting Ada memory management in the iAPX-432," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 117-131, 1982.

[38] J. Kaiser, E. Nett, and R. Kroger "Mutabor - An Intelligent Memory Management Unit for Object Oriented Auchitecture supporting Error Recovery," *Proceedings Fault-Tolerant Computing Systems 3rd International GI/ITG/GMA Conference*, 1987, pp. 61-71.

[39] J. Kaiser, E. Nett, and R. Kroger "MUTABOR - A Coprocessor Supporting Object-Oriented Memory Management and Error Recovery," *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, 1988, pp. 20-29.

[40] Jorg Kaiser, "MUTABOR, A Coprocessor Supporting Memory Management in an Object-Oriented Architecture," *IEEE Micro*, October 1988, 8(5) pp. 30-46.

[41] Umakishore Ramachandran and M. Yousef Amin Khalidi, "A Design of a Memory Management Unit for Object-base Systems," *Proceedings 1989 IEEE International Conference on Computer Design*, 1989, pp. 512-517.

[42] Umakishore Ramachandran and M. Yousef Amin Khalidi, "A Measurement-based Study of Hardware Support for Object Invocation," *Software-Practice and Experience*, September 1989, 19(9) pp. 809-828.

[43] M Yousef Amin Khalidi, *Hardware Support for Distributed Object-based Systems*, Ph. D. thesis, Georgia Institute of Technology, 1989.

[44] Gerry Kane, *MIPS R2000 Processor Architecture*, Prentice Hall, 1988.

[45] M. V. Wilkes, "Hardware Support for Memory Protection," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 1982, pp 107-116.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

[46] R. S. Fabry, "Capability-Based Addressing," *Communications of the ACM*, July 1974, 17(7), pp 403-412.

[47] *80386 Programmer's Reference Manual*, Intel, Inc., 1986.

[48] John L. Hennessy and David A. Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, 1989.

[49] John Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, August 1978, 21(8), pp 613-641.

[50] Keith Diefendorff and Michael Allen, "Organization of the Motorola 88110 Superscalar RISC Microprocessor," *IEEE Micro*, Vol. 12, No. 2, April 1992, pp. 40-63.

[51] *MC88110 Superscalar RISC Microprocessor User's Manual*, Motorola, Inc., 1992.

[52] Robin W. Edenfield, Michael G. Gallup, William B. Ledbetter, Jr., Ralph C. McGarity, Eric E. Quintana, and Russell A. Reininger, "The 68040 Processor: Part 2, Memory Design and Chip Verification," *IEEE Micro*, Vol. 10, No. 3, June 1990, pp. 22-35.

[53] *MC88110 RISC Instruction-Level Simulator Kit User's Guide*, Motorola, Inc. 1991.

[54] Michael J. Phillip, "Simulated Performance of the Motorola 88110 RISC Microprocessor," Internal Motorola Report, Motorola, Inc., 1991.

[55] *MVME197 Premier Performance Single Board Computer Data Sheet*, Motorola, Inc., 1993.

[56] Quentin Barnes, *MCGTools Bill of Materials*, Motorola internal software, Motorola, Inc., 1995.

[57] Gary S. Brown, *32-bit CRC*, 1986.

[58] Leif Lonnblad, Paul Rensing, Irwin Sheer, and Dag Bruck, *Class Library for High Energy Physics*, a collection of software, version 0.15, 1994.

[59] Howard D. Owens, *hsim - A Configurable Simulator for Evaluating MMU Performance*, Motorola internal software, Motorola, Inc., 1992.

[60] *OSE*, version 4.2b1, Dumpleton Software Consulting Pty Limited, 1995.

[61] Quentin Barnes, *TRS80 Simulator*, 1995.

[62] Nick Sayer, *Z-80 emulator*, 1985.

[63] Bill Dimm, *FeynDiagram*, version 2.2, [FTP: hepth.cornell.edu], 1993.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

[64] Jongjung Woo, *Hybrid Approach to Memory Organization*, Ph.D. Dissertation, The University of Texas at Austin, 1993.

[65] *Electronic Signals and Transmission Protocols, ISO/IEC 7816-3 Standard for Integrated Circuit Cards with Contacts*, International Organization for Standardization and the International Electrotechnical Commission, 1989.

[66] Henry Lieberman, "The Debugging Scandal and What to Do About It," *Communications of the ACM*, Vol. 40, No. 4, April, 1997, pp. 27-29.

[67] Mirowslaw Malek, Mohan Guruswamy, Mihir Pandya, and Howard Owens, "Serial and Parallel Simulated Annealing and Tabu Search Algorithms for the Traveling Salesman Problem," *Annals of Operations Research*, Vol. 21, 1989, pp 59-84.

[68] Howard Owens, "Computer Memory: Abacus to DRAM," *IEEE Potentials*, December, 1989, pp. 32-35.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

VITA

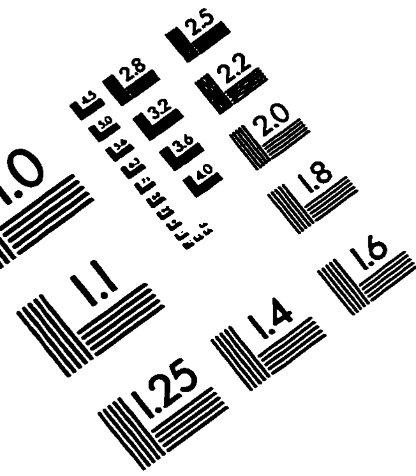
Howard Dewey Owens was born in Freeport, Texas, on June 28, 1958, the son of Herman Dewey Owens and Flossie Owens Hartman. After completing his work at Brazosport High School, Freeport, Texas, in 1976, he entered Brazosport College at Lake Jackson, Texas. In September, 1977, he transferred to Lamar University at Beaumont, Texas. In the summers of 1979, 1980, and 1981 he was employed by the Dow Chemical Company, Freeport, Texas. He received the degree of Bachelor of Science in Electrical Engineering from Lamar University in May, 1981. In September, 1981, he entered The Graduate School of The University of Texas and was employed as a teaching assistant for two semesters. He received the degree of Master of Science in Engineering from the Department of Electrical Engineering, The University of Texas at Austin, in August, 1983. Since that time he has been employed by Motorola, Inc., where his most recent position is Staff Engineer in the Advanced Computer Architecture Lab, Corporate Software Center. In January, 1987, he again entered the Graduate School at The University of Texas to continue his studies while remaining a Motorola researcher. His publications include "Serial and Parallel Simulated Annealing and Tabu Search Algorithms for the Traveling Salesman Problem" [67], "Computer Memory: Abacus to DRAM" [68], and "Software Defect Classification Using Purify" [3].

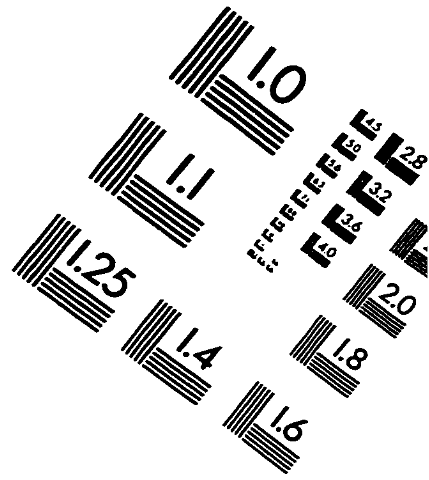
Permanent Address: 1808 Romeria Drive, Austin, Texas 78757

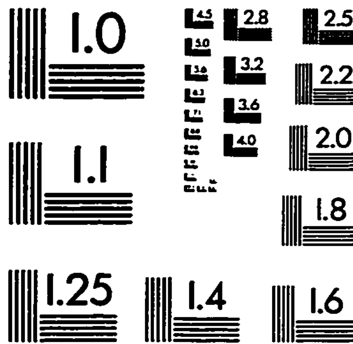
This dissertation was typed by the author using FrameMaker® on a Macintosh®.

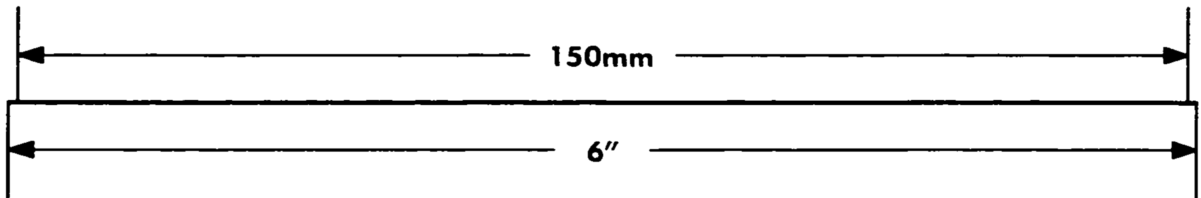
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com

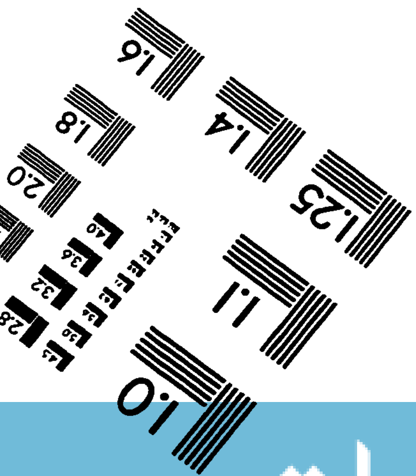
IMAGE EVALUATION TEST TARGET (QA-3)

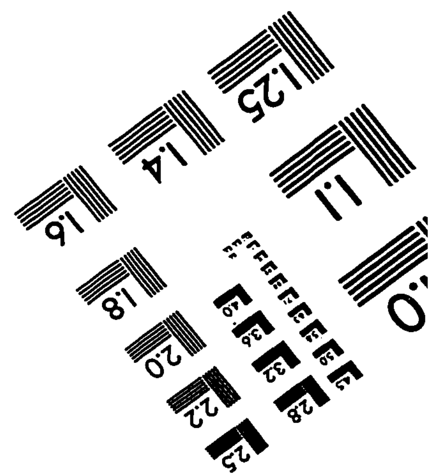












APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission. www.manaraa.com