SYSTEMATIC COMPUTER ARCHITECTURE PROTOTYPING

BY

THOMAS MARTIN CONTE

B.E.E., University of Delaware, 1986
M.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

SYSTEMATIC COMPUTER ARCHITECTURE PROTOTYPING

Thomas Martin Conte, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1992
Wen-Mei W. Hwu, Advisor

The current design process for workstation systems is over-taxed due to the size and diversity of realistic workload models. This thesis advocates a method to improve the design process by synthesizing prototype architectures from workloads. Focus is placed on the processor and memory systems, although the overall philosophy is applicable to other workstation components as well. Prototyping of memory systems is performed using methods to evaluate multiple designs with one pass over the address trace. Statistical sampling of address traces is adapted from traditional cache simulation to improve the performance and trace-size range of the techniques. These methods are extended to account for performance penalties due to multiprogramming. Prototyping of superscalar processors is performed using new statistical sampling techniques in conjunction with two simulation algorithms. The resource usage in an unlimited-resource simulation is used to select processor resource needs, but is only applicable to fixed operation latencies. A more general solution based on *simulated annealing* is developed and demonstrated as a prototyping method. The interaction between prototypes is investigated via simulation of three common schemes for coupling the processor to the memory system: *blocking* on a cache miss, *limited-blocking*, and *non-blocking*. It is concluded that sufficient design options exist to justify the separate design of the processor and the memory.

DEDICATION

Dedicated in fond memory to my grandfathers: Amato Conte and Joseph Giovanni.

ACKNOWLEDGEMENTS

I would like to thank my parents and sisters who have supported me with precious love and advice. My advisor, Professor Wen-mei W. Hwu, deserves special thanks for his guidance, insight and patience. I also thank the members of my committee who helped to shape both this work and this Ph.D. student. They are Professors W. Kent Fuchs, David J. Kuck, and Janak H. Patel. Thanks are also due to Professors Yale N. Patt and Edward S. Davidson of the University of Michigan who have supplied me with advice and encouragement.

Thanks go to my friends, of whom only a partial list is: Michael Chan, Chani, Gary Delp, Paul Domagala, Michael Golden, Marianne Holma, Bill Mangione-Smith, Jon Nadler, Isadora Parrini, Guru Parulkar, Brian Rogers, William Hsu, Eric Thayer, Robert Tomilson and Steve Turner. I would like to thank Sadun Anik and Scott Mahlke for many interesting discussions. I have had the pleasure of working in a research group composed of very bright people. Thanks to: Roger Bringmann, Pohua Chang, William Chen, Aloke Gupta, Grant Haab, Rick Hank, John Holm, Sabrina Hwu, Scott Mahlke, Nancy Warter and Tokuzo Kiyohara. Thanks also to Mrs. Thompson who helped me find the key.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

It is difficult to design successful workstation architectures since these systems are general-purpose and used for a large number of very diverse tasks. Performance of a workstation can be decomposed into that of its components, such as the network, graphics hardware, I/O, processor and memory subsystems. Successful design of these components requires careful consideration of the workload of workstation users. This workload is large and diverse. Current methods for designing these components are iterative processes that are not well-suited to large, diverse workloads. This thesis addresses this problem by developing a systematic method to synthesize prototype architectures of workstation components from large workloads. The thesis focuses on the processor and the memory components, although the overall approach can be applied to the evaluation of other workstation components as well. Special simulation techniques are developed for the task that are fast yet can tackle the very large design space of potential prototypes. New methods are developed to select prototypes from this design space based on cost and performance criteria. The assumption that the processor and memory system components

can be designed independently is checked empirically for three contemporary methods to interface the processor and memory system components together.

1.1 The Current Architectural Design Process

Current engineering design methods rely heavily on an iterative approach that begins with the selection of an initial architecture prototype. After the initial prototype is decided upon, it is used as the starting point to a process that evaluates the performance of the prototype and alters its design iteratively until an acceptable level of cost and performance is reached [1]. After the architectural design process is completed, computer-aided design tools are used to realize the architecture. The inputs to the architectural design process are test workloads, often industry-standard benchmarks or end-user supplied applications.

In an effort to better model the workloads of real users, the workloads used for design have grown in size and sophistication, and a general trend toward even larger and more-realistic workloads continues [2],[3]. At the same time, this increase in size places new burdens on the architectural design process, lengthening the process considerably. The simplistic solution to this problem is to limit the size of the workloads used, but this defeats the purpose of using a large workload and results in a poor match between architectural features and end-user needs.

1.2 Systematic Computer Architecture Prototyping

It is clear that there are considerable problems with the current architectural design process. What is needed is a systematic approach that can take workload elements such as industry-standard benchmarks or end-user supplied applications and derive architecture prototypes from them. The advantages of such a computer architecture prototyping technique would be:

- The architectural design process would begin with a prototype that is already influenced by diverse workloads,
- Since the architectural design process is iterative, improving the quality of the initial guess also improves the rate of convergence of the overall design process, reducing the number of iterations and enlarging the usable workload size.

This thesis presents a systematic approach to computer architecture prototyping with emphasis on cache-based memory systems and superscalar processor components of workstation systems.

1.2.1 The aims of computer architecture prototyping

The aims of computer architecture prototyping are:

- The selection of prototype designs from large workloads.
- The use of a large design space of possible designs.
- The ability to use realistic, large workloads.

These aims are also the challenges of computer architecture prototyping. In order to use a large design space, a general representation of workload behavior must be selected. Since architecture defines the interface between the programmer and the hardware, it is difficult to measure this interaction independent of architectural features. The approach taken in this thesis is to define an interface that includes many viable architectures.

A large design space may require simulation of each point in the design space. Simulation time of each design point is proportional to the length of the workloads used as inputs. Since improving the architectural design process involves reducing overall simulation time, the goals of using a large design space and the ability to use realistic, large workloads are at odds with each other. This thesis attacks the problem by proposing new simulation algorithms that simulate large design spaces very efficiently. To reduce the simulation time further, system cost models are introduced in the case of processor prototyping to reduce the design space to cost-effective designs.

1.2.2 The method of computer architecture prototyping

The method of computer architecture prototyping developed in this thesis begins by decomposing the workload into workload elements or *benchmarks*. The behavior of a benchmark is measured by executing the benchmark on an existing system and recording a *trace*: a record of the requests made by the benchmark of memory locations or instructions to execute. The trace is recorded with the aid of the compiler. (The format of the trace is discussed in Chapter 2.) The traces of a benchmark are used to

obtain performance of the memory system or the processor. This can be done while the benchmark runs or by using a previously generated and recorded trace.

The trace is used as input to simulations of the memory system. Simulation techniques are used that are capable of evaluating multiple cache memory designs with one scan through the trace. These *single-pass* techniques were first proposed by Mattson et al. [4]. The techniques are extended to include the effects of multiprogramming (context switching). The result of a single-pass simulation is the performance of a large space of possible cache memory designs. Examples are given of selecting prototypes from this space based on cache performance criteria.

Two simulation techniques are proposed to design prototypes of the processor, each with its own advantages and disadvantages. Two approaches to prototyping using the simulations are also proposed. In the first approach, the benchmark is simulated with an unlimited amount of resources. Criteria are developed to determine actual resource needs based on the usage of resources in the unlimited-resource case. These criteria are evaluated to determine their validity. In the second processor prototyping approach, the problem is formulated as an optimization problem where the parameters are the number, latency and cost of the function units. A design technique based on *simulated annealing* is proposed and demonstrated as a method to find near-optimal prototypes.

In order to evaluate large design spaces, the speed of simulation is crucial. Many benchmarks are too large to simulate in reasonable time. To address this problem, the trace of a benchmark is first reduced in size by use of statistical sampling techniques.

Sampling was first proposed by Laha et al. as a method to improve traditional cache memory simulation [5]. This thesis proposes new sampling techniques for the processor and singe-pass memory system simulation.

1.2.3 Further uses of computer architecture prototyping

Computer architecture prototyping can be used for two classes of problems that are separate from its application to the architectural design process. One such use is to characterize benchmarks and the other use is to aid in purchasing decisions.

Benchmarks are not only used to design new systems, they are also used to evaluate the performance of existing systems. The trouble with relying on benchmarks to evaluate existing systems is that the characteristics of the benchmarks themselves are unknown. For example, consider two workstations that are identical in every respect except one workstation has a high-performance memory system and the other has a low-performance memory system. A benchmark that requires only a small amount of memory will not be able to distinguish between these two systems. If the purchaser who is trying to interpret the benchmark results has in-house applications that require large amounts of memory, the purchaser should not use this particular benchmark to decide which system to buy.

Computer architecture prototyping can help the above situation by determining the memory system needs for each benchmark and every in-house application. In these situations, the architecture prototypes are also guides to benchmark characteristics. Purchasers should only trust benchmark results from those with characteristics that match

the characteristics of the end users' in-house applications. The idea of using prototypes for benchmark characterization was suggested by Conte and Hwu [6].

The characterization of in-house applications can be used for purchasing decisions in another way. After characterizing in-house applications, the buyers can match the characteristics to those of existing systems in the marketplace. The buyers can then be assured that the purchased system will have an architecture that matches the needs of their in-house applications.

This thesis emphasizes the challenges of generating prototypes and does not consider benchmark characterization or purchasing in further detail. For further discussion, see [6].

1.3 Relation to Previous Work

Methods for *hierarchical performance evaluation*, first proposed by Kumar and Davidson in [7], are similar to computer architecture prototyping. The aims of hierarchical performance evaluation is to use a hierarchy of performance models with a cost function to select cost-effective designs. The top levels of the hierarchy are occupied by methods that are time-efficient yet have low performance estimating accuracy. These are used to narrow the design space. This narrower design space is then searched using less time-efficient, higher accuracy methods.

Where hierarchical methods seek to replace the design process, computer architecture prototyping is a method to improve the first step. Each approach has its own advantages.

The advantages of generating prototypes are the two additional uses for prototypes, characterizing the benchmarks and aiding in purchasing decisions.

Linear performance models sum the effects of classes of operations to arrive at an estimate of the total run time of a benchmark on a system [8],[9],[10]. For example, the sum of the dynamic count of floating-point operations times the time for a floating-point operation would produce the portion of a program's run time that was due to floating-point multiplication. In general, the performance is recreated as a sum of the benchmark's frequency of use of an operation times the system's latency for executing the operation. The limitations of linear performance models are that they do not account for non-linear effects due to parallelism or for saved state between operations (such as in cache-based memory systems). Modern workstation designs exploit available parallelism and preserve state between operations to achieve high performance. Because of this, linear performance models are inappropriate for the prototyping of workstation designs.

Single-pass cache simulation is a method to evaluate multiple cache designs using one pass over the address trace [4]. Single-pass techniques are ideally suited for the purposes of prototyping and are a topic of this thesis. Statistical sampling has been proposed as a method to speed up traditional cache simulation of very long address traces [5],[11]. Statistical sampling is adapted in this thesis for both single-pass memory system simulation and processor simulation.

The effects of multiprogramming on cache memory performance can invalidate design decisions made using uniprocessor cache simulations [12]. Multiprogramming effects have

been studied by many researchers [13] –[19]. However, no previous study has extended single-pass techniques to model multiprogramming effects. This thesis presents new single-pass techniques that take multiprogramming effects into account.

1.4 Organization of Thesis

This thesis is organized as follows. Chapter 2 presents a discussion of the workloads used and the design and collection of traces of the behavior of these workloads. Chapter 3 discusses the prototyping of cache-based memory systems with emphasis on single-pass simulation techniques. A new simulation technique that simulates multiprogramming effects is presented in this chapter. The application of statistical sampling is investigated as a method to improve the performance and trace-size range of memory simulation.

Chapter 4 discusses the architecture and prototyping of superscalar processors. This chapter introduces two simulation techniques for superscalar processor designs. Statistical sampling is applied to these techniques as in Chapter 3. These techniques are combined with cost functions and near-optimal search methods to select superscalar processor prototypes.

The interaction between the processor and the memory system is the topic of Chapter 5. The primary interaction is between the processor and the cache. When a cache miss occurs, the processor can take one of three strategies: block completely, perform limited blocking, or queue the miss requests using a non-blocking cache design. The three

methods are discussed and evaluated along with their respective impacts on prototype selection.

Conclusions and further discussions of this research and its contributions are presented in the final chapter of the thesis, Chapter 6.

2. BENCHMARKS AND INSTRUMENTATION METHODS

The workloads used to design computer architectures take many forms. They are typically composed of *benchmarks*, test workloads used to evaluate computer performance. The benchmarks are believed to be either representative of the intended workload of the system or capable of providing insight into the range of performance variation of the system. Investigation of the usage of benchmarks used by papers presented at the *International Symposia on Computer Architecture*, years 1984-1990, reveals seven common categories of benchmarks:

Application benchmarks are runs of programs that are widely used to perform some critical function for a user. Example application benchmarks are sessions from database systems or runs of numerical packages performing functions such as matrix multiply or Gaussian elimination.

Kernel benchmarks are code fragments extracted from real programs, where the code fragment is believed to be responsible for most of the execution time of the program.

Many of these benchmarks have the same advantages as the synthetic benchmarks (see below): small code size and long execution time. Kernel benchmarks appear to be heavily used in computer architecture studies [20].

Partial benchmarks are partial traces of programs. One drawback of their use is that independent reproduction of results is difficult since the the portion of the benchmark traced is often unknown.

Recursive algorithm benchmarks are programs implementing recursive algorithms such as the solutions to the Hanoi towers problem and the Nine-Queens problem. These benchmarks are not applications, they do not perform often-used functions.

Synthetic benchmarks are small programs especially constructed for benchmarking purposes. They do not perform any useful computation. The intent of synthetic benchmarks is to use a small program to approximate the average characteristics of real programs. At one time, the popularity of synthetic benchmarks was on the rise; however, they have experienced declining use in recent years.

OS utility benchmarks are taken from commonly used utilities such as the Unix `sort` or `tar`.

Unspecified benchmarks is a catch-all category for unspecified workload descriptions. Some uniprocessor papers used traces collected from systems in which the use of the system was either not reported in the paper or unknown.

Table 2.1: Benchmark usage in *International Symposium on Computer Architecture*, 1984-1990.

	Year							
Category	1984	1985	1986	1987	1988	1989	1990	Percentage
Applications	2	2	2	5	4	3	5	16%
Kernel	3	2	6	4	5	3	4	19%
Local	1	1	4	0	2	3	7	13%
Partial	0	0	2	0	2	2	1	5%
Recursive	3	5	6	5	3	6	4	22%
Synthetic	1	0	3	5	1	0	0	7%
OS Utility	1	2	1	4	2	4	5	13%
Unspecified	2	2	1	0	1	1	0	5%

Local benchmarks are programs that are site-specific or in-house applications that are not widely available (in contrast to application benchmarks that are widely used and available).

Use of unspecified and partial benchmarks occurs relatively infrequently in the papers that were surveyed. Table 2.1 presents the count of papers using benchmarks from each category (note that in some instances one paper used benchmarks from several categories). A complete list of the papers and their categorization are presented in [20].

In 1989, the System Performance Evaluation Cooperative released version 1 of a benchmark suite designed to evaluate the performance of workstations [3]. This benchmark suite is now commonly referred to as SPEC89. Table 2.2 lists benchmarks that are members of the SPEC89 suite (compiled with aid of [3]). The table also assigns one of the above categories to each benchmark.

Table 2.2: The SPEC89 benchmark suite.

Benchmark	Categories	Description
doduc	application	Monte Carlo simulation of the time evolution of a thermohydraulical modelization for a nuclear reactor.
eqntott	application	Generates truth table from logic equations.
espresso	application	Performs PLA optimization.
fpppp	application	Simulates two electron integral derivative (quantum chemistry).
gcc	OS utility	GNU C compiler, version 1.35.
matrix300	kernel	Performs 300×300 matrix multiply.
nasa7	kernel, synthetic	Seven kernels: a matrix multiply, an FFT, a Cholesky decomposition, a tradiagonal matrix solution, a Gaussian elimination, a boundary value problem, and a three-matrix pentadiagonals inversion (developed by NASA Ames).
spice2g6	application	Performs analog circuit simulation (with greycode digital circuit as input).
tomcatv	kernel	Generation of finite mesh.
xlisp	application, recursive	Lisp interpreter (the application) executing the Nine-Queens problem (the recursive benchmark).

The SPEC89 benchmark suite covers six of the eight categories of benchmarks presented above (the remaining two categories, unspecified and partial, have negative connotations). The suite is used in this thesis as a workload to test the computer architecture prototyping methods that are developed herein. Its use does not constitute a claim that the suite is representative of the workloads of actual workstations. However, since the benchmark suite is being used to evaluate the performance of existing workstations, it is the logical choice for the design of future workstations [6]. Since SPEC89 has this role in future designs, computer architecture prototypes derived from SPEC89 benchmarks serve as important tests of the prototyping techniques.

The following sections discuss measurement of the behavior of benchmarks such as members of the SPEC89 benchmark suite in some detail.

2.1 Collection of Traces of Benchmark Behavior

A benchmark as it executes makes requests of the system and alters its behavior accordingly. The requests can be in the form of instructions to execute, references to memory locations, requests to the kernel for system functions, etc. The term *trace* is used here to refer to a log of these requests.

2.1.1 Address trace collection

Figure 2.1 shows a memory hierarchy with one cache indexed by virtual addresses, one cache indexed by physical addresses, the physical memory and the virtual memory

of the system. Only the address usage is shown in the figure, the data path has been omitted. An address trace can be taken at any point along this hierarchy. Address

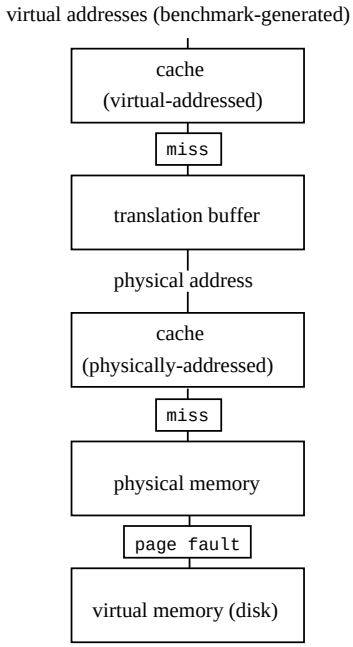

Figure 2.1: A typical memory hierarchy showing address usage.

traces taken lower in the hierarchy are specific to the behavior of the combination of the program and the hardware. For example, in the figure the physical cache in the picture receives only references that miss in the virtual cache and are translated to physical addresses; therefore, its address trace is a function of the design of the virtual cache. It is also a function of the multiprogramming level of the system, since the virtual-to-physical page mapping depends on the ensemble of the address requests of all jobs running in the system. However, pages are only mapped to physical locations at power-of-two boundaries. If the trace is composed of the virtual addresses, the simulation of a first-level virtual cache is valid, and the simulation of a first-level physical cache with size less

than the page size is also valid. For the remainder of the hierarchy, a prototype cache for a given level can be used to *filter* the trace for use in simulating the next-level cache. For these reasons, the *virtual address trace*, a trace of the virtual addresses referenced during the execution of the program, is used in this thesis.

The compiler can aid in tracing the virtual address trace by generating extra instructions surrounding all load/store operations. These added instructions record the addresses of the load or store them in a *trace buffer*. During program execution, the trace buffer is periodically flushed to a trace consumer (e.g, a simulator) or written to a file. This technique is an application of *software instrumentation* and has been used by Larus for *AE*, Stunkel and Fuchs for *TRAPEDS*, and Golden for *Spike*, among others [21]–[23]. This thesis uses Spike for software instrumentation of virtual address traces.

2.1.2 Instruction trace collection

Capturing traces of instructions poses a difficult problem for a system created to prototype hardware. Instruction encodings are idiosyncratic and vary widely from vendor to vendor. In the case of some RISC architectures, the instruction set reflects the latencies of the underlying hardware (e.g., the number of branch and load delay slots) [24]. This further complication causes the traces recorded using some instruction sets to be hardware specific.

This thesis uses the GNU C retargetable, optimizing compiler (version 1.40) with optimizations enabled to translate benchmark source into executables [25]. The intermediate

code of a retargetable compiler is designed to be translated into many different instruction sets. The instruction-set architecture of the processor in this thesis is assumed to be this intermediate-code language. This approach has been used previously in [26]. The processor simulation methods rely only on the type of each instruction and the identities of the operands. For this reason, other instruction formats can be used with the trace without modification. The use of intermediate-code removes machine-specific operand restrictions or instruction-format restrictions. Delay slots and other hardware-specific features are not encoded. The resulting trace is free of anomalies that might complicate the interpretation of the results.

The traces of the intermediate code are collected by creating templates of the intermediate instructions for each basic block at compile time. Also at compile time, additional code is added to mark the entrance of each basic block and record its *id* in the trace buffer. At run time, the contents of the trace buffer are interpreted and the *id*s of basic blocks are used in conjunction with the database of intermediate-instruction templates to generate a trace of the intermediate instructions. This instruction trace appears to the trace consumer as though the benchmark had executed on a machine with the intermediate code as its instruction set. Further discussions of this technique are presented in [22] and [23].

2.1.3 In-process trace generation

Traces are traditionally written to a secondary storage device and then used for simulation. Two of the SPEC89 benchmarks, gcc and espresso, have data address traces of slightly more than 33 million and 150 million references, respectively. Each of the references is a 32-bit quantity, resulting in sizes of 132–600 megabytes of storage. The gcc and espresso benchmarks have the two smallest data address traces of the benchmarks in the SPEC89 benchmark set. This suggests that trace storage conflicts with the aim of architecture prototyping to use large, diverse workloads. Statistical techniques can be used to reduce the size of traces to manageable sizes [5]. This thesis further develops these ideas. In order to validate the results of the statistical techniques, the simulation results from full-trace simulations are still required.

This thesis uses a solution to the problem of large trace size termed *in-process trace generation*. In this technique, the simulation is combined with the benchmark in the same process. The simulator is compiled as an auxiliary function to the benchmark, and the compiler inserts periodic calls to the simulator to flush the trace buffer. The trace need not be recorded since it is re-generated by running the benchmark. Section 3.2.2 of Chapter 3 exploits this trace generation method to improve upon statistical techniques.

2.1.4 Library code

The most common operating systems used for workstations are based on 4.3BSD Unix [27]. Unix employs a set of user-code libraries to implement many functions. Exclusion of library code from the trace would exclude frequently used functions. To circumvent this problem, software-instrumented versions of the libraries are compiled and used along with the benchmarks. The specific libraries are taken from SunOS version 4.1.1, but generic versions of most library functions are used.

In addition to library functions, the operating system kernel interacts with the benchmark either at the request of the benchmark via system calls or periodically to implement multiprogramming [27]. This thesis treats these system call interactions as special events and records their occurrence. As an example of system call usage, Table 2.3 reports the

Table 2.3: Dynamic system call usage.

Benchmark	System calls
gcc	`getrusage (6072), write (164), sbrk (66), read (15), open (2), close (2), fstat (2), ioctl (2), getpagesize (1)`
espresso	`write (27), sbrk (18), read (4), ioctl (2), fstat (2), open (1), getpagesize (1)`
spice	`write (784), lseek (6), fstat (5), sbrk (3), close (3), read (3), ioctl (3), getpagesize (1)`
xlisp	`sbrk (7), ioctl (2), fstat (2), open (1), getpagesize (1), read (1)`
matrix300	`lseek (12), fstat (5), close (5), stat (3), sbrk (3), write (3), ioctl (2), open (2), getdtablesize (1), getpagesize (1)`

system calls used by some members of the SPEC benchmark set and the number of dynamic occurrences of each system call (in parentheses). The `getrusage` system call that is prominent for gcc is used by the `times` library call to report run times. The `sbrk` call is used in the heap space allocator. Heap space is used significantly in xlisp and espresso. Since the `malloc()` function is included in the tracing process, heap management is included in the virtual address trace. I/O intensive benchmarks reveal themselves here by the use of the `read, write, open, close, fstat, lseek,` and `ioctl` calls. These benchmarks are spice and gcc. The dynamic percentage of execution taken up by system call events was not reported in Table 2.3 since it would be close to zero for all cases.

Kernel code is specific to each vendor and as such is difficult to generalize to a system-independent format. However, it is possible to view the kernel as simply another benchmark for the purposes of prototyping. The kernel appears to the hardware as just another program that it must execute along with other programs (benchmarks) in the system. Although this is an appealing approach to the treatment of prototyping using the kernel, the kernel is not included in the benchmark set used in this thesis. System call interactions between an executing benchmark and the kernel are captured, and interactions such as interrupts due to expired quanta are modeled. These models are used to adjust the prototypes of the memory system in Chapter 3.

3. MEMORY SYSTEM PROTOTYPING

Multiple levels of caching and buffering have become the norm in memory system design. Memory hierarchies composed of cache memories are so crucial to high-performance computer architecture design that performance evaluation of cache memories has received phenomenal attention. In 1991, Smith catalogued 487 technical papers and reports that dealt with some aspect of caching [28]. This chapter addresses the problem of deriving prototypes of memory systems designed with caching. To do this requires measurement of the performance of a large number of cache designs. This cache performance evaluation process must be fast yet accurate. The importance of accuracy is self-evident. A fast method is important so that memory address traces from long-running benchmarks can be used to explore a large design space of potential prototypes.

Researchers have devised analytic models and novel simulation approaches to measure cache performance [4],[29],[30]. Analytic cache models achieve low accuracy and are therefore inappropriate for prototyping memory systems. Of the simulation approaches, the direct approach is to simulate the cache at the register-transfer level. This approach

is called the *traditional cache simulation* approach throughout this chapter. Prototyping demands simulation of a large number of cache designs, limiting the usefulness of traditional cache simulation. To eliminate the number of required simulations, *single-pass cache simulation* is used. Such methods simulate multiple cache dimensions in a single pass by exploiting the inclusion property of stacking replacement algorithms (*least-recently used* is the most common member of this class of replacement algorithms [4]). This method has been extended to include rigid placement/replacement algorithms used in direct-mapped caches [31]. Single-pass cache simulation is ideally suited for prototyping.

This chapter focuses on using the recurrence/conflict single-pass cache simulation technique for prototyping memory systems. Two extensions to this single-pass technique are discussed. The speed of the technique is improved while maintaining acceptable accuracy by adapting statistical sampling techniques developed for traditional simulation to work with single-pass simulation. The single-pass technique is also extended to capture multiprogramming effects. Multiprogramming degrades memory system performance since (process) context switching reduces the effectiveness of cache memories. For the prototypes to be correct, multiprogramming effects must be taken into account.

3.1 Single-Pass Cache Simulation

A traditional cache simulator uses a data structure that is a replica of the tag store of the cache being simulated. The simulation involves updating this data structure at

each reference. When an address in the trace is not present in the tag store structure, the corresponding cache miss is recorded. The advantages of such a technique are its efficiency and simplicity. A simple array can be used for the tag store of a direct-mapped cache. The time complexity for such an algorithm is $O(N)$ in N inputs. Since the tag store does not change in size during simulation, the space complexity is $O(1)$.

The disadvantage of the traditional cache simulator is its lack of generality. A simulation must be performed for each configuration of cache under study. Hence the term *multiple-pass* cache simulator can be used to describe the traditional simulator since it requires multiple passes over the trace.

Single-pass cache simulation techniques rely on the inclusion property of stacking replacement algorithms. Exploitation of this property allows this class of simulators to find the miss ratios for an entire design space of cache dimensions with one pass over the trace. The space complexity of these algorithms is directly proportional to the *static* program size. Hence, it is $O(1)$. The disadvantage of these approaches is their time complexity, which is $O(N \times d)$, where d is the average stack depth [30]. The particular single-pass simulation approach presented in this paper is based on the recurrence/conflict model of the miss ratio. The model is introduced below followed by a description of the simulation method.

3.1.1 Recurrences and conflicts

The metric used in many memory system studies is the miss ratio. This is the ratio of the number of references that are not satisfied (i.e., that *miss*) for a cache at a level of the memory system hierarchy over the total number of references made at that level. The miss ratio has served as a good metric for memory systems since it is a characteristic of the workload (e.g., the memory trace) yet independent of the access time of the memory elements. A given miss ratio can be used to decide whether a potential memory element technology will meet the required access time for the memory system [13]. The recurrence/conflict model of the miss ratio is best illustrated with an example. Consider the trace of Figure 3.1. The *recurrences* in the trace are accesses e, f, g and h. In the ideal case of an infinite cache, the miss ratio may be expressed as

$$\rho = \frac{N - R}{N}, \tag{3.1}$$

where R is the total number of recurrences and N is the total number of references. Non-ideal behavior occurs due to *conflicts*. A *dimensional conflict* is defined as an event which converts a recurrence into a miss due to limited cache capacity or mapping inflexibility. For illustration, consider a direct mapped cache composed of two one-byte blocks shown in Figure 3.2. (Note that in practice, such a small cache would be impractical to build.)

Reference	a	b	c	d	e	f	g	h
Address	0	1	2	3	1	2	1	0

Figure 3.1: An example trace of addresses.

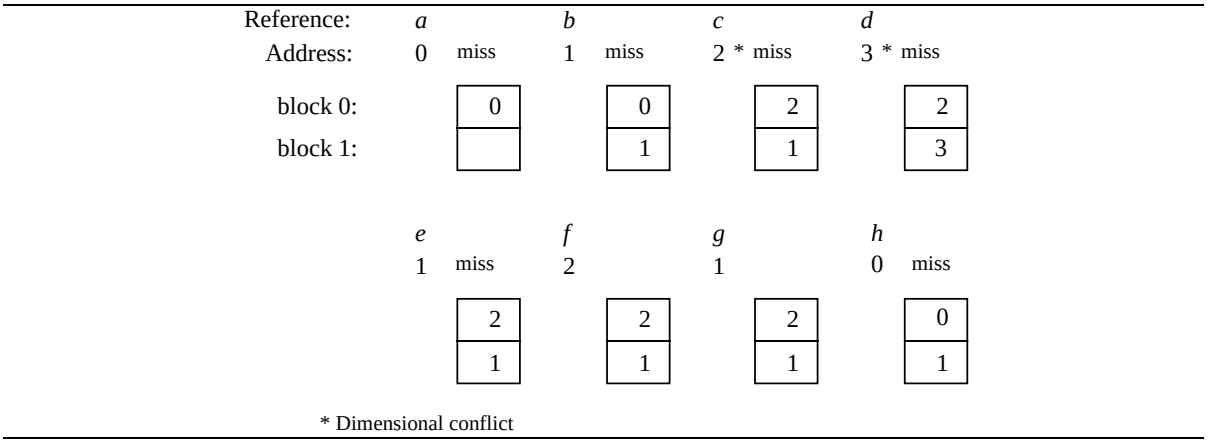

Figure 3.2: An example two-block direct-mapped cache behavior.

A miss occurs for the recurring reference e because reference d purges address 1 from the cache due to insufficient cache capacity. Similarly, a miss occurs for recurring reference h due to reference c. References d and c represent a dimensional conflict for the recurrences e and h, respectively. The other misses, a, b, c and d, occur because these are the first references to addresses $0, 1, 2$ and 3, respectively. The following formula can be used for deriving cache miss ratio, ρ, for a given trace, a given cache dimension:

$$\rho = \frac{N - (R - D)}{N}, \tag{3.2}$$

where D the total number of dimensional conflicts. (For the example, $\rho = (8 - (4 - 2))/8 = 0.75$.) This is a general model and can be extended to account for other effects, such as conflicts due to multiprocessor cache coherence [32] and context switching (see Section 3.3).

3.1.2 Reference streams and cache dimensions

A formal abstraction of a benchmark's trace is termed a *reference stream*. This is a sequence of references to addresses, $w(k)$, of length N $(0 \leq k < N)$. When required, the addresses are represented by lower-case Greek letters, such as α, β, γ. The reference stream is assumed to be generated by a single process in a multiprogramming system. Note that a reference at $w(k)$ occurs later than $w(k+1)$ in time, but the parameter k does not represent parameterized time since it does not take into account the difference in service times between cache hits and cache misses. For this reason, k is referred to as the *reference count*. The trace also contains information about voluntary context switching. A reference is called a *voluntary context-switch event* if the benchmark relinquished the CPU after the reference (e.g., a system call was performed).

The dimension of a cache is expressed using the notation, (C, B, S), for a cache of size 2^C bytes, with block size 2^B bytes, and 2^S blocks contained in each associativity set. The term *set size* is used to mean associativity level, or the number of blocks per set. *Cache size* is the total number of bytes per cache. *Block size* has been called *line size* elsewhere [17]. Note that $C \geq B + S$. The notation (C, B, ∞) is an abbreviation for the dimension of a fully associative cache $(S = C - B)$. For example, a cache of dimension $(10, 6, 0)$ is a 1KB direct-mapped cache with a block size of 64 bytes; and, a cache of dimension $(21, 10, 11)$ (alternately, $(21, 10, \infty)$) is of size 2MB with 1KB-length blocks and it is fully associative. A dash is substituted for an entry in the triple to indicate all caches of that dimension: $(-, 5, 1)$ are all caches with block size of 32 bytes and having

2-way associativity. Caches are assumed to use LRU replacement and map addresses into sets using bit selection [30].

It is useful to partition the reference stream by setting the block offset portion of all addresses in the stream to zero. This produces a *block reference stream*, $w_B(k)$, which is defined such that

$$w_B(k) = 2^B \left\lfloor \frac{w(k)}{2^B} \right\rfloor .$$

In binary, this is equivalent to setting the least-significant B bits to zero.

3.1.3 Least-recently used (LRU) stack operation

Least-recently used (LRU) stacks were first introduced by Mattson et al. in [4] as a way to model the behavior of paging systems. An LRU stack operates as follows: when an address, $w_B(k) = \alpha$, is encountered in the block reference stream, the LRU stack is checked to see if α is present on the stack. If α is not present, it is pushed onto the stack. However, if α is present (e.g, it is a recurring reference), it is removed from the stack,

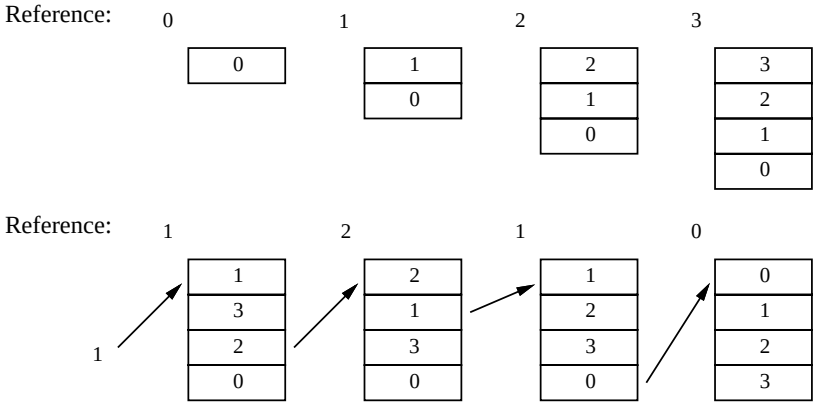

Figure 3.3: An example of LRU stack operation.

then repushed onto the stack. This is illustrated in Figure 3.3 for the example reference stream at the beginning of this section (Figure 3.1).

A stack is represented as $S_B(k)$, maintained for a block size B at time k. The ith ordered item of $S_B(k)$ is expressed as, $S_B(k)[i]$. The stack may also be expressed as an ordered list, such that $S_B(k) = \{S_B(k)[0], S_B(k)[1], \ldots, S_B(k)[m]\}$, where m is the depth of the stack. The following operations are defined for a stack:

the **push**$(\cdot)$ function,

$$\textbf{push}(S_B(k), \alpha) = \Big\{ \alpha,\ S_B(k)[0],\ S_B(k)[1],\ \ldots,\ S_B(k)[m] \Big\},$$

the $\boldsymbol{\Delta}(\cdot)$ function,

$$\Delta(S_B(k), \alpha) = i, \qquad \text{if } S_B(k)[i] = \alpha,$$

and, the **repush**$(\cdot)$ function,

$$\begin{aligned} \textbf{repush}(S_B(k), \alpha) \ = \ & \Big\{ \alpha,\ S_B(k)[0],\ S_B(k)[1],\ \ldots,\ S_B(k)[\Delta(S_B(k), \alpha) - 1], \\ & S_B(k)[\Delta(S_B(k), \alpha) + 1],\ \ldots,\ S_B(k)[m] \Big\}. \end{aligned}$$

$\Delta(S_B(k), \alpha)$ and **repush**$(S_B(k), \alpha)$ are undefined when $\alpha \notin S_B(k)$. When $S_B(k)$ and α are understood, it is convenient to use $\boldsymbol{\Delta} = \boldsymbol{\Delta}(S_B(k), \alpha)$. Note that **push**$(\cdot)$ and **repush**$(\cdot)$ are defined as side-effect-free functions rather than procedures. This is to remove dependence on the time variable, k.

The least-recently used management policy for a stack is shown in Figure 3.4 for an address $\alpha = w_B(k)$. In Step 1.1, the references between the top of stack and the recurring

reference have been referred to as the set , $= \{\beta_i \mid \beta_i = S_B(k-1)[i], 0 \leq i \leq \mathbf{\Delta}\}$. Figure 3.4 is applied to $\alpha = w_B(k)$ for all k. The LRU policy is essentially a definition

```
1.      if α ∈ S_B(k − 1) then
1.1         do_recurrence(α, , )
1.2         S_B(k) ← repush(S_B(k − 1), α),
2.      else S_B(k) ← push(S_B(k − 1), α)
3.      N ← N + 1
```

Figure 3.4: The least-recently used management policy for a stack, $S_B(k)$ (adapted from Mattson *et al.*).

for calculating $S_B(k)$ from $S_B(k-1)$ and α. In most situations, $S_B(k)$ is calculated in order to obtain other statistics, such as the stack depth distribution.

The complexity of the algorithm of Figure 3.4 depends on the complexity of the `do_recurrence()` procedure. Assume for the moment that the complexity of this procedure is $O(d)$ on average (the validity of this assumption is justified below). The outer algorithm's complexity is also dependent on the efficiency of the set (stack) existence operator in Step 1. In Mattson *et al.* [4], the set-existence operation was determined by scanning the entire stack. This has an average complexity $O(d)$ for the set existence, where d is the average stack depth [30]. This results in a complexity of $O(N \times d)$ for the entire algorithm. This approach to calculating set existence can be replaced by using a hash table lookup, where each entry of the new table contains a pointer to the stack frame. Hash table lookup also has complexity of $O(d)$ on average [33]. However, there is a practical advantage to using hash table lookup. For the hash table implementation,

only the hash conflict set for the block needs to be searched to determine whether the reference is first-time. This is a constant-time improvement and does not change the asymptotic behavior.

3.1.4 Recurrence/conflict-based single-pass simulation

The single-pass cache simulation algorithm for limited associativities ($S < C - B$) is created by expanding the `do_recurrence` procedure of Figure 3.4 [30]. A single-pass algorithm that uses the recurrence/conflict model is presented in Figure 3.5. This algorithm is similar to the original algorithm of Traiger and Slutz [31]. However, where Traiger and Slutz recorded temporal localities, this algorithm records recurrences and conflicts. Since temporal locality functions can occupy considerable space, using recurrences and conflicts is an advantage. In this respect, the recurrence/conflict approach is similar to the algorithm of Hill and Smith [30].

Whenever a reference is found on the stack in Figure 3.5, its presence indicates that it is a recurrence. The calculation of the number of recurrences ($R[B]$) is implemented by recording the number of times this event occurs in Step 1 of Figure 3.5. The remainder of the algorithm is devoted to calculating the dimensional conflicts ($D[C, B, S]$).

The **for** statement that iterates for all intervening references in Step 2 of Figure 3.5 calculates the raw information for determining two classes of cache organizations. The maintenance of the number of unique references (u) in Step 2.1 is used to calculate the largest-sized fully associative cache with a dimensional conflict (c_∞). This calculation is

```
do_recurrence(α, , ):
1       R[B] ← R[B] + 1
2       for β_i ∈ ,  do
2.1         u ← u + 1
2.2         d ← |β_i − α|
2.3         z ← ctz(d)
2.4         p[z] ← p[z] + 1
2.5         z_max ← max(z, z_max)
3       c_∞ ← ⌊lg u⌋ + B
4       D[c_∞, B, ∞] ← D[c_∞, B, ∞] + 1
5       z ← z_max
6       S_target ← 1
7       nss ← 0
8       for s ← 0 to S_max
8.1         C_MC ← B
8.2         while z ≥ 0 and nss < S_target
8.2.1           nss ← p[z]
8.2.2           z ← z − 1
8.3         if nss ≥ S_target then
8.3.1           C_MC ← z + s + 1
8.4         D[C_MC, B, s] ← D[C_MC, B, s] + 1
8.5         S_target ← 2 × S_target
```

Figure 3.5: The recurrence/conflict single-pass cache simulation algorithm.

done in Steps 3 and 4 by finding the lg (log base two) of this count. Unlike the algorithm of Mattson *et al.*, only cache sizes that are multiples of powers of 2 are considered [4]. The remainder of Step 2 calculates a histogram, $p[z]$, of a function of the current reference (α) and each intervening reference (β_i) (Step 2.4). This function is the lowest power of two factor of the arithmetic difference between the two references (Steps 2.2 and 2.3). For a range of direct-mapped caches, this function is equivalent to the largest cache size in which a miss will still occur for α due to the intervening reference to β_i. The remainder of the procedure uses this information to calculate this cache size for all associativities (Steps 5–8).

The histogram ($p[z]$) is processed for all associativities by scanning the histogram from largest to smallest potential conflicting cache size. A set size can be thought of as a *conflict tolerance*. A conflict between α and β_i for a direct-mapped cache of dimension $(C, B, 0)$ is equivalent to α and β_i occupying the same set in caches (C, B, S) for $C - B \geq S > 0$. The larger the set size, S, the more numerous the allowed same-set mappings between references to α before these mappings result in a miss.

In Steps 6 to 8, the set sizes are considered in increasing order to determine how many same-set mappings are tolerable. For each set, the largest cache size in which a miss will occur, C_{MC}, (MC = *maximum conflict*) is the product of the same cache size for a direct-mapped cache times the set size (Step 8.3.1, note that addition of these exponents of base 2 implies multiplication). If no same-set mappings remain in the histogram,

the only conflicts accounted for are those that occur in caches containing a single block (Step 8.1).

The complexity of the inner-most **while** statement of Steps 8.2–8.2.2 is dependent on S_{target} and z. The initial value of z is $z_{\max}$, which is bounded by the word size of the trace (e.g., 32 bits) since $z_{\max}$ is an indirect result of the $\mathrm{ctz}(d)$ function of Step 2.3. Therefore, z has a constant upper bound. The other determiner of the **while** statement's execution is S_{target} which is 2^s due to Steps 6, and 8.5. Therefore, S_{target} also has a constant upper bound of $2^{S_{\max}}$. The **while** statement therefore has a worst-case execution time of $\max(z_{\max}, 2^{S_{\max}})$, which results in a complexity of $O(1)$. The surrounding **for** statement also has a complexity of $O(1)$, resulting in a total complexity for Step 8 and all of its substeps of $O(1)$.

The complexity of the `do_recurrence()` procedure is $O(d)$ in the worst case due to the scanning of the stack in Steps 2.1–2.5. An input that elicits worst-case behavior is a cyclic referencing pattern of addresses, such as

$$\alpha, \beta, \gamma, \delta, \alpha, \beta, \gamma, \delta, \alpha, \ldots,$$

where each cycle consists of $\alpha, \beta, \gamma, \delta$. Consider a trace of such a pattern of length N having K cycles. In such a trace, any recurrence must traverse $d = N/K$ references in Steps 2.1–2.5. Hence, the complexity of this is $O(N/K)$, or simply $O(d)$.

3.2 Statistical Sampling of Address Traces

Cache simulation is concerned with summarizing the cache performance due to the entire trace by reporting a small set of statistics. By and far the most common statistic is the miss ratio; its decomposition into recurrences and conflicts was discussed above. Since the miss ratio is an arithmetic average over time, it can be accurately predicted by statistically sampling the trace [5]. Statistical sampling has been applied to traditional cache simulation, but it has not yet been applied to single-pass techniques. Such an extension is developed here.

3.2.1 Definition of sampling

Consider a reference stream, $w_B(k)$. Statistical sampling takes N_S samples of length L_S from this trace. Note that each sample is a *contiguous* block of L_S references. Typical values for N_S are 40 samples and typical values for L_S are in the range of 50,000–200,000 references [5],[34]. The removed references between two samples comprise a *sample gap* that has length L_G. Without loss of generality, it is assumed that L_G is a constant and that L_S and L_G are selected so that the trace is an integral multiple of $L_S + L_G$ (i.e., $N = N_S(L_S + L_G)$). These samples are applied to the cache simulator in the order they are taken from the trace. The state of the cache is unknown between each sample. An issue in sampling for cache systems is how to repair the state of the cache between the application of each sample to achieve high accuracy. Several approaches to the repair of the cache state are described below.

In the following discussion, the actual miss ratio is denoted ρ and the miss ratio estimated by sampling is denoted ρ'. This convention is used for the components of the miss ratio, $R[B]$, $D[C,B,S]$ and N. The sampled miss ratio is, therefore,

$$\rho' = \frac{N' - (R'[B] - D'[C,B,S])}{N'}. \tag{3.3}$$

There are several ways of measuring the error between ρ and ρ'. The percentage change, or *relative error*, can be calculated as $\mathrm{RE}(\rho) = |\rho - \rho'|/\rho$. The problem with the use of relative error with miss ratios is that it may inflate differences between ρ and ρ' that would not normally matter to a designer. Designers typically pick a target miss ratio value and accept any design that satisfies that value [6]. In particular, a cache with a miss ratio of $\rho = 0.01\%$ is as acceptable for a design as a cache with a sampled miss ratio of $\rho' = 0.005\%$, since both values are very small. However, the relative error between these values is $\mathrm{RE} = 50\%$. The lesson contained in this example is that although relative error is appropriate for large values of the miss ratio, it is less significant for smaller values. This suggests that an error measure weighted by the actual miss ratio would better characterize the impact of the difference. Such a measure is the *absolute miss ratio error*, $\mathrm{AE}(\rho) = |\rho - \rho'|$. This section uses absolute error measures for comparison between sampling approaches.

3.2.2 State repair techniques

The cache state repair problem for statistical sampling of address traces is how to accurately estimate the contents of the cache at the beginning of each sample. Several

approaches have been proposed for traditional cache simulation [5],[11],[35],[34]. This section adapts one of these approaches to single-pass techniques and then presents a new approach. The two single-pass approaches to state repair are compared using empirical simulation results. Because the SPEC89 benchmarks are long-running, only the shortest of the benchmarks can be used for these comparisons, since the result from a full simulation is required to evaluate sampling. The shortest two benchmarks in the SPEC89 set are gcc and espresso. Even these two benchmarks have large traces: the full gcc benchmark address trace is slightly more than 33 million references long, and the trace for espresso is slightly more than 150 million references long. To increase the diversity of this benchmark set, the first 100 million references from four additional SPEC89 benchmarks are used to evaluate sampling. These truncated benchmarks are taken from doduc, nasa7, spice and xlisp.

The *fill-flush* approaches to state repair entail removal of all unknown unique references from the trace [35]. To apply this approach to single-pass simulation, the recurrence/conflict single-pass method is extended to measure the number of references whose states are unknown due to lost state between samples. It is traditional to term these references *fill references* [35]. Let $F[B]$ be the count of fill references for block size B. In the adapted algorithm, the cache is flushed between each sample and $F[B]$ is collected. References with unknown state are removed from calculation of the miss ratio by using Equation (3.4):

$$\rho = \frac{N - (R[B] - D[C, B, S])}{N - F[B]}. \tag{3.4}$$

Figure 3.6 shows the recurrence/conflict single-pass algorithm modified for this form of sampling. Flushing of the LRU stack is represented as "$S_B \leftarrow \emptyset$" (Step 1.1 of Figure 3.6). Step 1.2.1 shows the sampled references being removed by indexed calculation of the reference stream, w_B. This need not be the case, instead the samples could be written to disk or consumed in-process. Equation (3.4) can be re-arranged as follows:

$$\rho = \frac{N - (R[B] - D[C, B, S] - \rho F[B])}{N}, \tag{3.5}$$

(see [35]). Equation (3.5) shows that removing the fill references is equivalent to weighting these references by the miss ratio of the remainder of the trace.

```
1.          for i ← 0 to N_S − 1
1.1             S_B(0) ← ∅
1.2             for j ← 0 to L_S
1.2.1               α ← w_B(i × (L_S + L_G) + j)
1.2.2               if α ∈ S_B(j − 1) then
1.2.2.1                 do_recurrence(α, , )
1.2.2.2                 S_B(j) ← repush(S_B(j − 1), α),
1.2.3               else
1.2.3.1                 S_B(j) ← push(S_B(j − 1), α)
1.2.3.2                 F[B] ← F[B] + 1
1.2.4               N ← N + 1
```

Figure 3.6: Extension of the recurrence/conflict single-pass algorithm to sampling using fill-flush state repair.

Typical values for absolute error are presented in Figures 3.7–3.12 for direct-mapped and fully associative caches with sample sizes of $L_S = 50,000$, 100,000, and 200,000 references. The relatively large error for nasa7 is a consequence of fixed-interval sampling (i.e., constant L_G). Two trends can be observed in the figures:

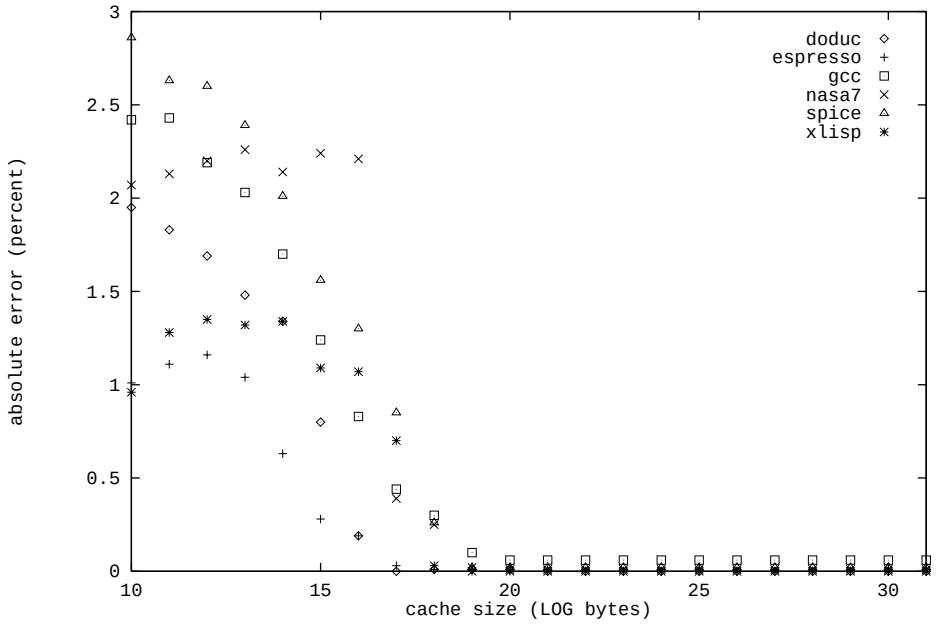

Figure 3.7: Absolute error for fill-flush, direct-mapped caches, $L_S = 50,000$.

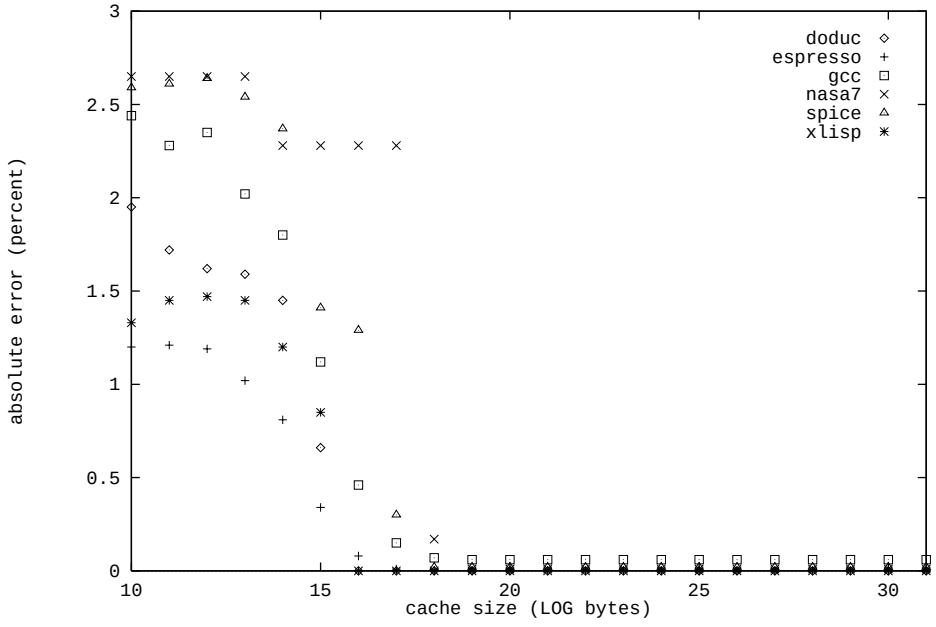

Figure 3.8: Absolute error for fill-flush, fully associative caches, $L_S = 50,000$.

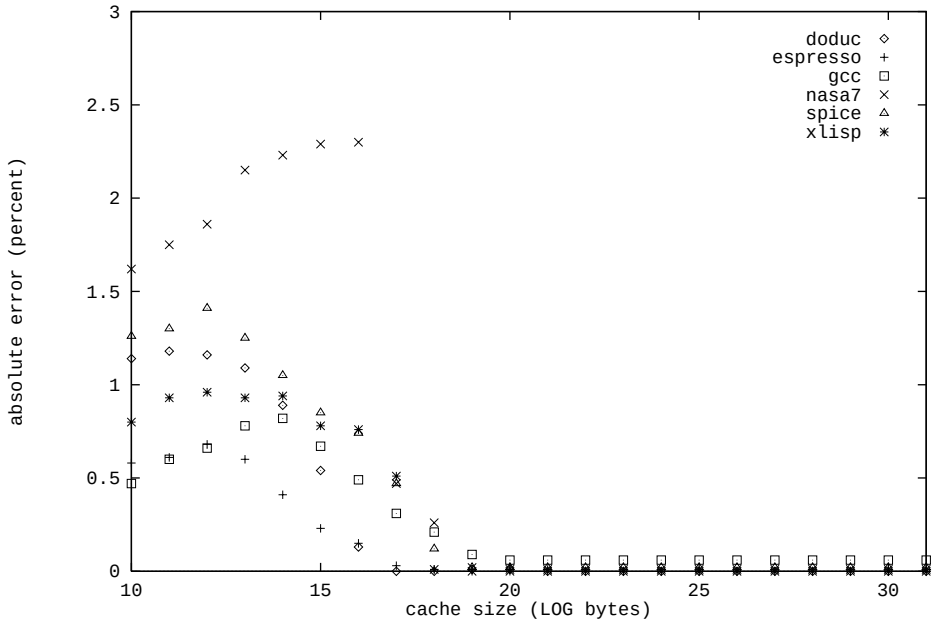

Figure 3.9: Absolute error for fill-flush, direct-mapped caches, $L_S = 100,000$.

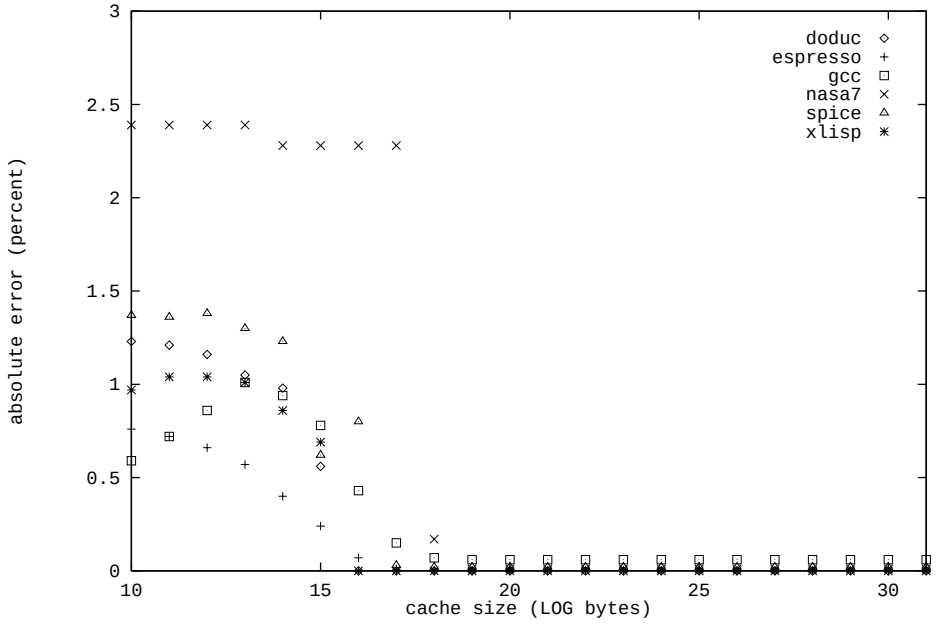

Figure 3.10: Absolute error for fill-flush, fully associative caches, $L_S = 100,000$.

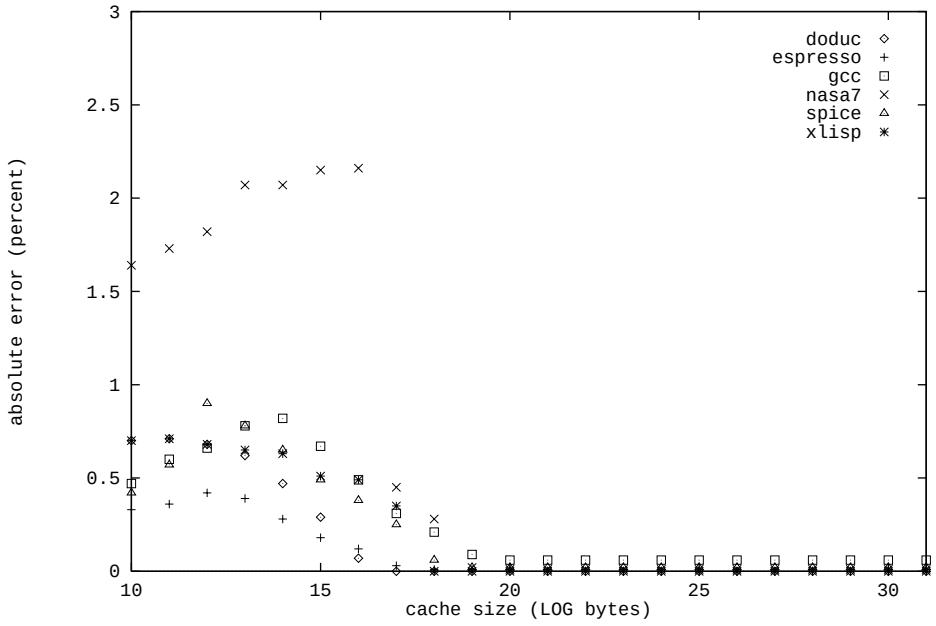

Figure 3.11: Absolute error for fill-flush, direct-mapped caches, $L_S = 200,000$.

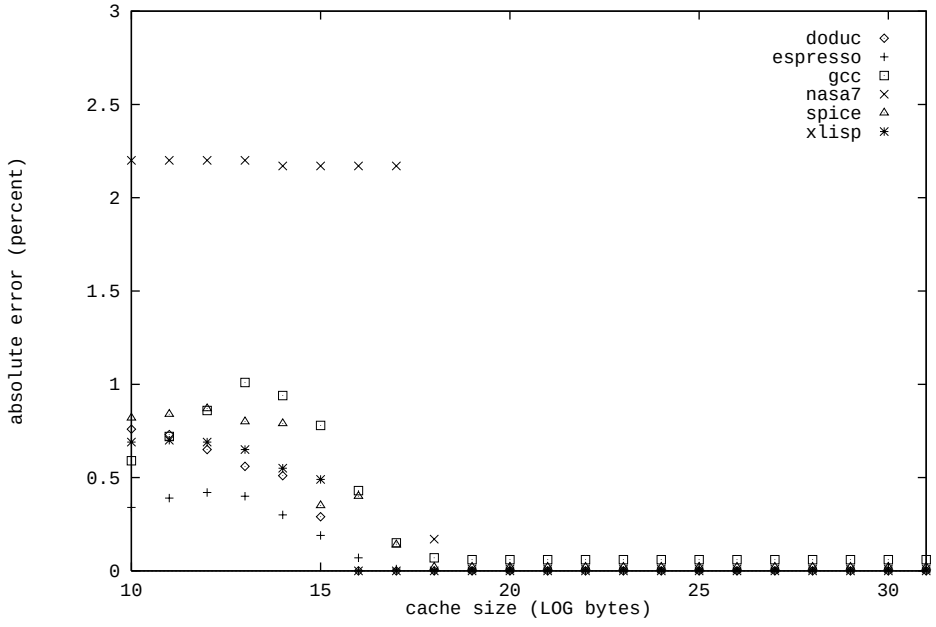

Figure 3.12: Absolute error for fill-flush, fully associative caches, $L_S = 200,000$.

1. Absolute error decreases with sample size.

2. There is a constant absolute error (approximately 0.06%) for large cache sizes across all sample sizes.

Observation 1 is due to increasing sample size which results in the simulation of larger numbers of references from the original trace. Observation 2 is due to the fact that the fill references can potentially contain first-time references to locations. When C is increased to the point where $D[C, B, S] = 0$, then,

$$\rho(C \text{ sufficiently large}) = (N - R[B])/N. \tag{3.6}$$

However,

$$R[B] = R'[B] + f_r \times F[B], \tag{3.7}$$

where f_r is the fraction of fill references that recur. The fraction f_r can only be known by measuring the entire trace since the status of fill references is unknown due to state loss. Assuming that all fill references are recurrences effectively sets $f_r = 1$, resulting in the constant error observed in the figures. An approach to removing this error is discussed below.

The ratio of sample simulation time for the full-flush approach to the non-sampled approach (i.e., using the full trace) for the gcc and espresso benchmarks is presented in Figure 3.13. This ratio can be thought of as the *speedup* of the sampling approach. Small sample sizes obtain a higher speedup over large sample sizes in the figure since

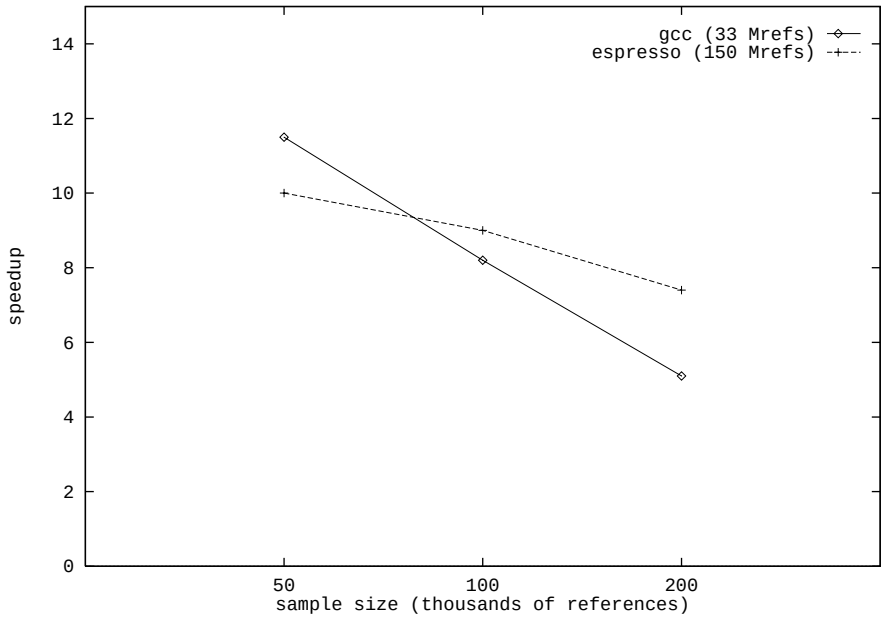

Figure 3.13: Speedup of fill-flush approach over no sampling for benchmarks gcc and espresso.

sampled traces with smaller sample sizes are shorter overall, resulting in less work for the simulator.

There are several other approaches to state repair that can collectively be termed *cache warm-up* approaches. In these approaches, calculations of performance metrics are delayed for each sample until the cache contents are stable, or "warmed up." This warm-up state is specified by some criterion, where the criterion vary based on the approach. One possible warm-up criterion is to use a fraction of the sample to prime the cache and then record recurrences and conflicts for the second half of the sample. This approach might work well for small caches using traditional simulation. It does not perform well

for larger caches using traditional simulation and as such does not perform well for single-pass methods. Several other warm-up criteria are

1. Delay calculation of performance metrics for each set in the cache until each set is filled with references [11],

2. Redefine the miss ratio in terms of the lifetime of references and measure expected lifetime length [34].

Method 1 is not applicable to single-pass cache simulation since the state of every associativity set in every possible cache dimension is not kept. The state could be kept. However, since the maintenance of this information is equivalent to the maintenance of state information for traditional simulation, the algorithm would reduce to traditional simulation. Method 2 also requires maintenance of additional state information, in this case to maintain the lifetime of each reference in the LRU stack. Lifetime is specific to a particular cache configuration and 2 has similar problems to 1 with respect to large data structures. Instead of using 2 to extend single-pass methods to achieve lower absolute error than the fill-flush techniques, an approach that avoids altogether the problem of state repair is possible.

In a situation in which in-process trace generation is occurring (see Section 2.1.3 of Chapter 2), the entire trace is available even if sampling is occurring at the input to the cache simulator. It is possible to make use of these excluded references. Consider Figures 3.4 and 3.5 from the previous section. The shorter LRU stack maintenance algorithm of Figure 3.4 could be applied to all the references in the trace, saving the

the more complicated and costly application of `do_recurrence()` of Figure 3.5 for the references inside the sample. In this approach, statistical sampling is used for the conflict metrics only and $R[B]$ and N are recorded for the whole trace. The miss ratio is then calculated as follows,

$$\rho = 1 - \frac{R[B]}{N} - \frac{D'[C, B, S]}{N_S L_S}. \tag{3.8}$$

The power of this approach is twofold. First, if the simpler algorithm exploits a hash table for stack blocks, it can be made to run much faster than the more complicated algorithm that performs the `do_recurrence()` procedure. Second, since the stack is maintained, it is know whether any reference recurs for all references inside the sample. Hence, the *state of all references is known.* This approach is therefore termed a *no-state-loss sampling technique.*

The modified sampling algorithm is shown in Figure 3.14, where the predicate `sampling` tests if reference $\alpha = w_B(k)$ falls inside a sample. Using the notation for sample length and sample gap, this predicate is defined as

$$\texttt{sampling}(k) : k \bmod (L_S + L_G) \leq L_S. \tag{3.9}$$

To demonstrate the accuracy of the approach, the absolute errors are presented in Figures 3.15–3.20. Comparison of Figures 3.15–3.20 to Figures 3.7–3.12 reveals several advantages of the no-state-loss approach over the fill-flush approach:

1. **if** $\alpha \in S_B(k-1)$ **then**
1.1 **if** `sampling`(k) **then** `do_recurrence`$(\alpha, ,)$
1.2 $S_B(k) \leftarrow$ **repush**$(S_B(k-1), \alpha)$,
2. **else** $S_B(k) \leftarrow$ **push**$(S_B(k-1), \alpha)$
3 **if** `sampling`(k) **then** $N \leftarrow N + 1$

Figure 3.14: A no-state-loss approach to extending a single-pass cache simulation algorithm for sampling.

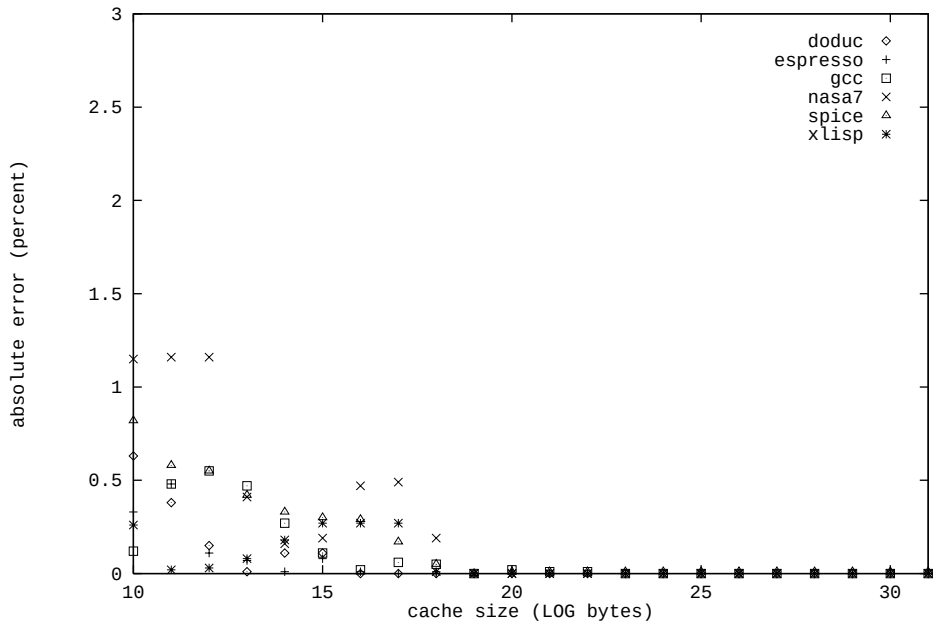

Figure 3.15: Absolute error for no-state-repair, direct-mapped caches, $L_S = 50,000$.

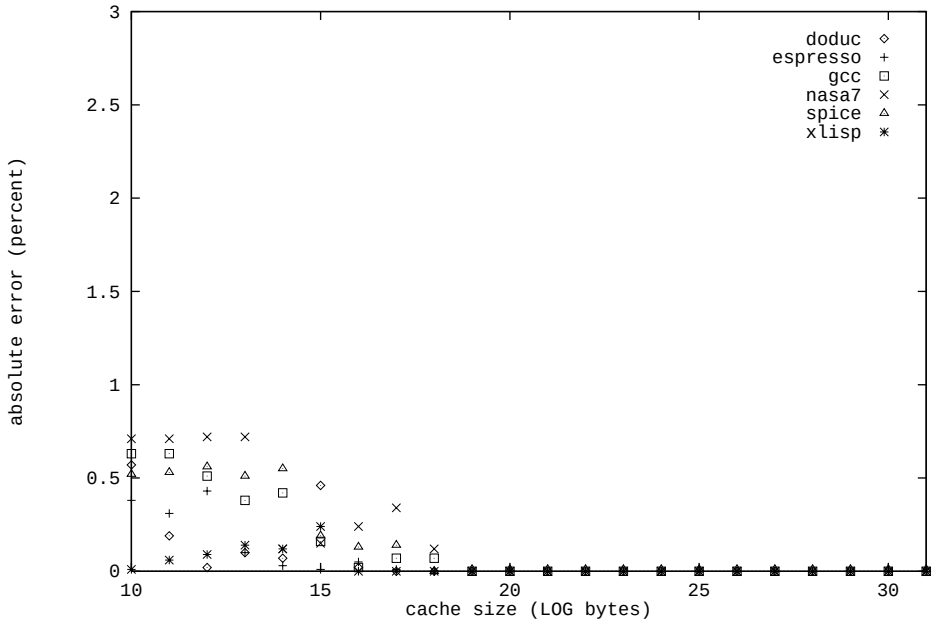

Figure 3.16: Absolute error for no-state-repair, fully associative caches, $L_S = 50,000$.

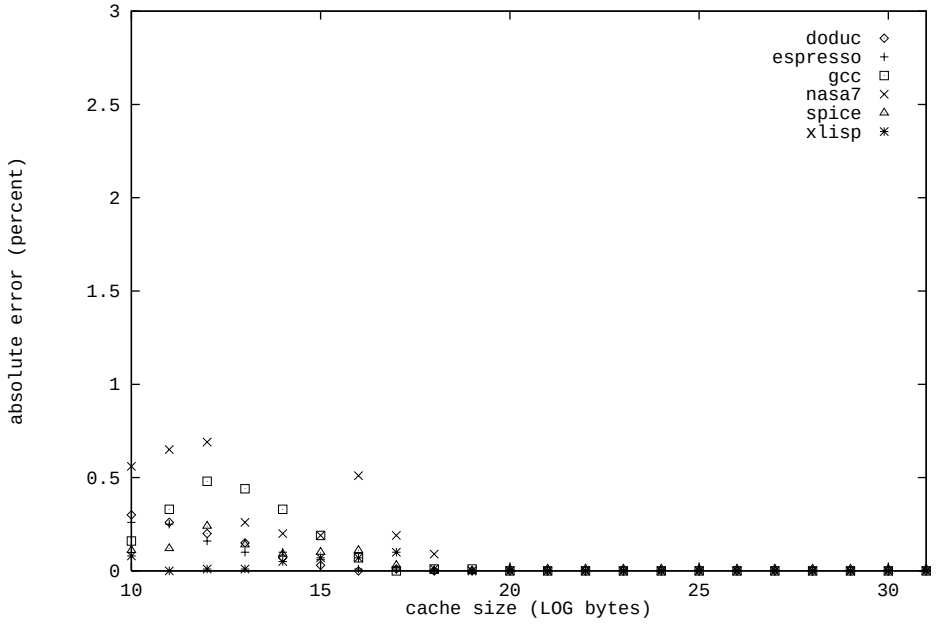

Figure 3.17: Absolute error for no-state-repair, direct-mapped caches, $L_S = 100,000$.

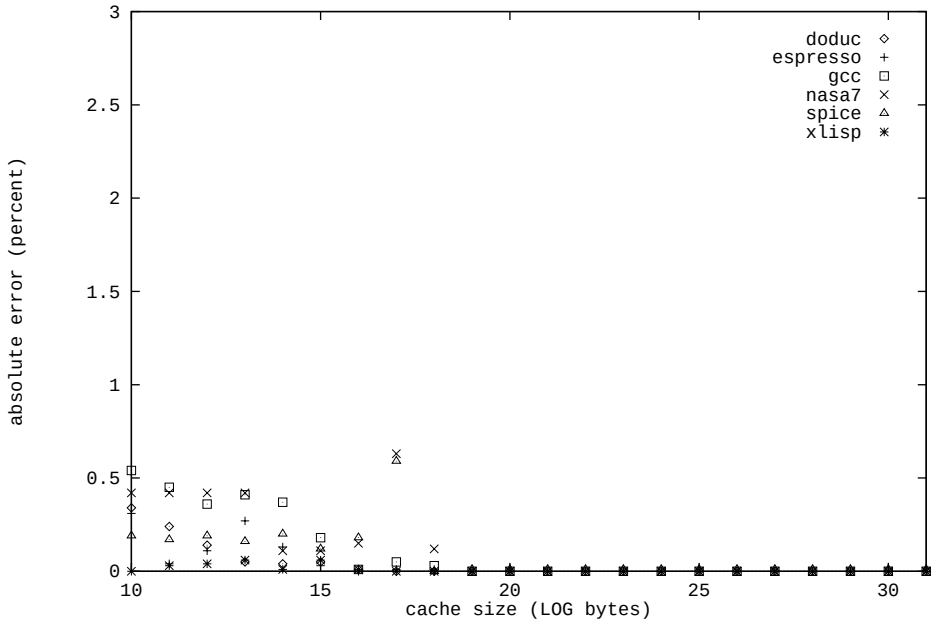

Figure 3.18: Absolute error for no-state-repair, fully associative caches, $L_S = 100,000$.

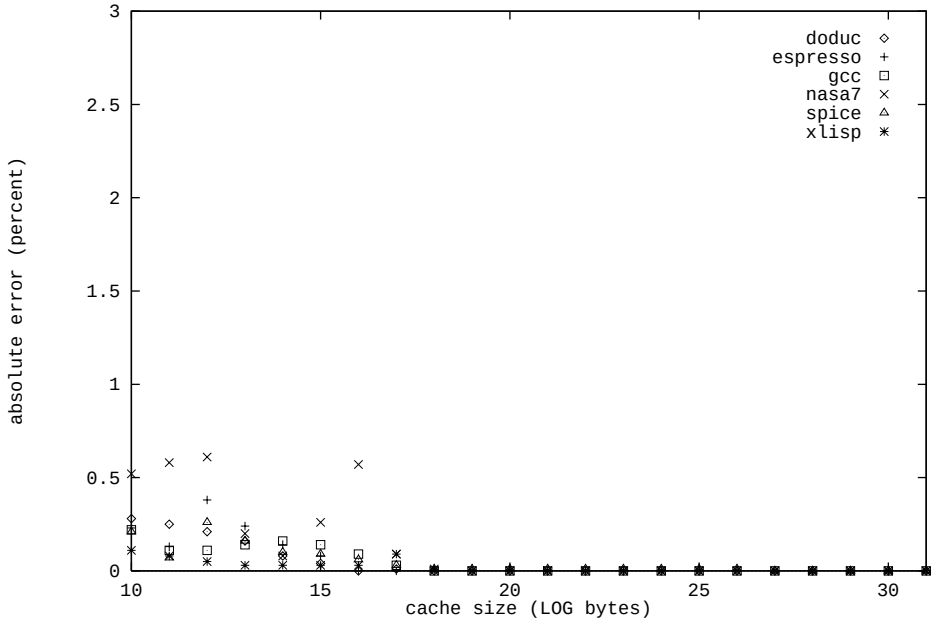

Figure 3.19: Absolute error for no-state-repair, direct-mapped caches, $L_S = 200,000$.

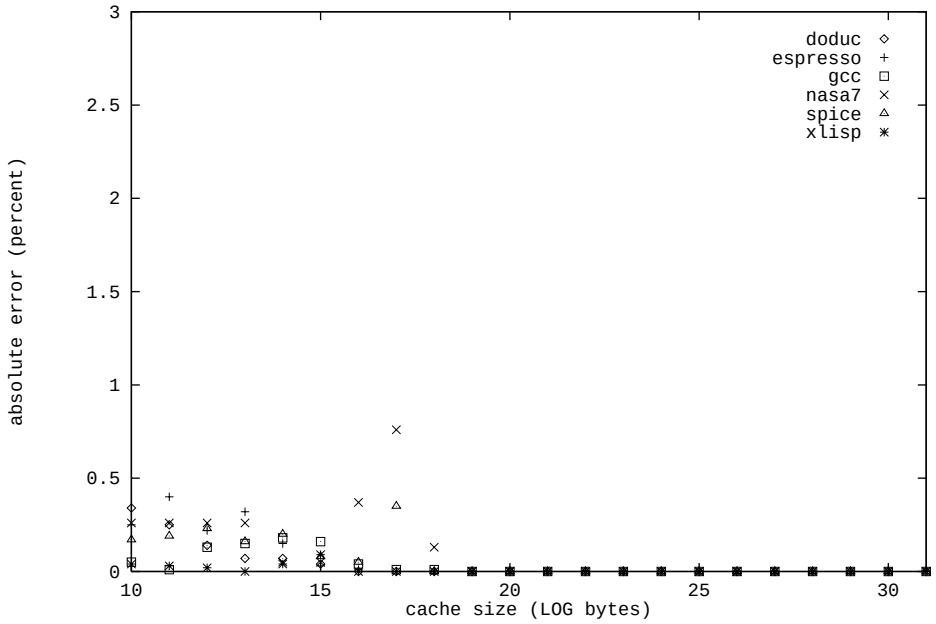

Figure 3.20: Absolute error for no-state-repair, fully associative caches, $L_S = 200,000$.

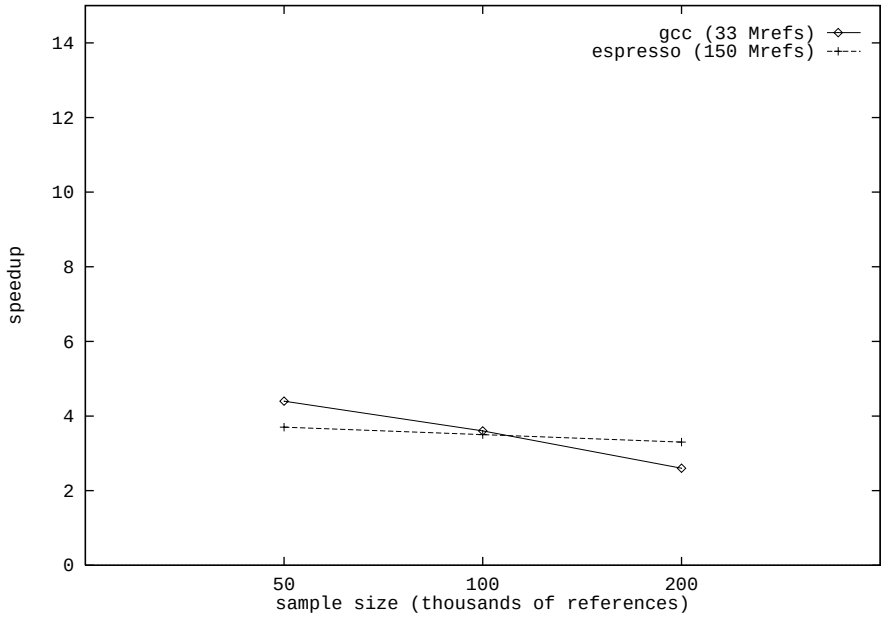

Figure 3.21: Speedup of no-state-loss approach over no sampling for benchmarks gcc and espresso.

1. In general, the error for no-state-loss is approximately half that of fill-flush.

2. The error is zero for large caches ($C > 19$, empirically) for no-state-loss compared to the constant error observed for fill-flush.

Both advantages 1 and 2 can be attributed to the full-trace measurement of $R[B]$. In particular, advantage 2 is a direct result of Equations (3.6) and (3.7) since f_r is known for no-state-loss.

A qualification must be made concerning this approach. It is not as fast as simple sampling techniques such as the fill-flush approach. This is the case since some processing must be done for all references. Figure 3.21 shows the speedup of no-state-loss over full-trace simulation for the gcc and espresso benchmarks, with the scale adjusted to match the scale of Figure 3.13. Comparison of Figure 3.21 to Figure 3.13 reveals that no-state-loss sampling is approximately 2-3 times slower than fill-flush sampling. Hence, fill-flush might be preferable when the benchmarks are extremely long-running and a factor of 2-3 is important. The higher error for medium-sized caches of the fill-flush approach suggests that the fill-flush approach is not well-suited for prototyping of first-level cache designs. The no-state-loss approach has higher accuracy than the fill-flush approach for all cache sizes and is generally preferable because of this property.

3.3 Susceptibility of Programs to Context Switching

Multiprogramming degrades memory system performance since (process) context switching reduces the effectiveness of cache memories. This occurs when cache contents that will be needed after the process returns from a context switch are purged by the intervening processes. The cache contents that may fall victim to context switching are determined by the process' reference pattern (a program characteristic) and the cache dimension (a system design parameter). The portion of such cache contents that are actually purged by intervening processes is determined by the load of the system: the number of ready processes and access patterns of these processes. The method presented in this section accurately records, for all cache dimensions and all context switching intensities in a single pass, the total amount of cache contents that will be needed after the process returns. This information is defined as the *susceptibility* of the program to the effect of context switching.

Several other approaches have been taken to measure the effects of context switching [13]–[19]. The earliest approaches flushed the cache being simulated at fixed intervals in the trace [13],[14]. Shedler and Slutz [15] approached the problem by stochastically merging several memory reference traces. Easton [16] used the average working set size of the memory reference trace to estimate cold-start miss ratios. Haikala [18] simplified Easton's approach by estimated cold-start miss ratios using a Markov chain model. Cold-start miss ratios can be used to approximate the multiprogramming effects. Switching between multiple memory reference traces at a fixed interval was used by Smith [17]

to measure multiprogramming effects. Also, hardware measurements of real multiprogrammed workloads were performed by Clark [12] and Agarwal et al. [19]. Apart from the approximations of Easton [16] and Haikala [18], no work has been done to extend single-pass methods to exactly model the effects of context switching. Since multiprogramming effects can account for a 4% to 12% degradation in performance [12],[18],[19], this omission in the literature has limited the usefulness of single-pass methods.

One obvious extension to single-pass methods to model context switching effects is to flush the LRU stack periodically. The shortcoming of this approach is that one simulation would have to be performed for each context switching intensity (e.g., time quantum and I/O workload). A more desirable method is to record the context switching effects for all intensities in one pass. This section introduces a single-pass method for measuring the susceptibility of a program to the effects of context switching for all cache dimensions and all intensities. It is demonstrated that the susceptibility measures can be combined with system load parameters and context switching intensity to yield the performance degradation in various multiprogramming environments without resimulation. Obtaining memory system performance degradation under many different system loads allows the memory system to be designed with a degree of robustness. It further increases the advantage of single-pass stack methods over multiple-pass methods. This is the first such study to make the dichotomy between program susceptibility and multiprogramming effects. The measured performance is compared to results from periodic and random flushings of the LRU stack.

3.3.1 Types of context switching

Context switching occurs due to two distinct events: (1) a *voluntary context switch*, where the benchmark relinquishes the processor, and (2) an *involuntary context switch*, where the benchmark's execution is suspended due to external interrupts. Voluntary context switches are a characteristic of the benchmark. They occur at the same place in the execution between different benchmark runs. On the other hand, involuntary context switches are determined by the I/O system behavior (device interrupts), clock frequency (timer interrupts), etc. They do not occur at the same place between runs of the benchmark and are not characteristic of the benchmark. Page faults are treated as involuntary context switches because page faults depend on the interaction of processes in the system, whose interaction is assumed to be pseudo-random in nature. Furthermore, the frequency of page faults can be reduced in prototypes by determining the correct size of physical memory to achieve a very low paging rate [6].

Since involuntary context switches occur at random instances, it is assumed that involuntary context switches can occur with equal probability for each reference in the reference stream [18]. This probability is denoted, q, and termed the involuntary *context switching intensity*. Separation of the system's characteristics from these characteristics of the benchmark allows many different systems to be considered without re-simulating the benchmark's behavior. This is the main goal of single-pass techniques in general [31]. Although the occurrence of involuntary context switches is not a characteristic of the benchmark, the benchmark's susceptibility to their occurrence is. This susceptibility

can be measured as the expected number of multiprogramming conflicts due to random involuntary context switching. A method to measure this susceptibility is presented in Section 3.3.3 that records the benchmark's susceptibility to all context-switching intensities in a single pass through the trace. The empirical results discussed in Section 3.4.1 demonstrate the validity of this single-pass approach.

The working set of a process (benchmark) may have been flushed from the cache before it re-enters the run state after a context switch. Let f_{CS} represent the *fraction of the cache's contents flushed between context switches.* The number of processes executed before a process returns from a context switch is a function of the system load and the operating system scheduling policy. Furthermore, the particular cache blocks flushed due to a context switch also depend on the reference patterns of the processes executing on the system. This makes f_{CS} highly dependent on several volatile variables and, therefore, difficult to measure. (Nevertheless, some empirical estimates of f_{CS} are presented in Section 3.4.5.) Some virtual memory system implementations force a cache flush to eliminate problems with page sharing of writable pages [19]. Also, it has been shown that for small cache sizes, a context switch effectively flushes the cache; therefore, $f_{CS} = 1$ [17]. For larger caches, this provides an upper bound for the effects of context switching.

3.3.2 The components of multiprogramming conflicts

Multiprogramming conflicts are defined in terms of *potential victims.* A recurring reference that is not removed from a specific cache by a dimensional conflict, yet that

may be removed by a context switch, is a potential victim of the context switch. Potential victims are defined as $X_V[C,B,S]$ and $\bar{X}_I[C,B,S,q]$, for all voluntary and involuntary context switches, respectively. $X_V[C,B,S]$ is the total number of potential victims due to voluntary context switching for caches of dimension (C,B,S). $\bar{X}_I[C,B,S,q]$ is the expected number of potential victims due to involuntary context switching of intensity q. The multiprogramming conflicts are expressed in terms of victims as

$$M[C,B,S,q] \equiv f_{CS}\left(X_V[C,B,S] + \bar{X}_I[C,B,S,q]\right). \tag{3.10}$$

The equation for the miss ratio (Equation (3.2)) can be modified to take into account the new conflicts,

$$\rho = \frac{N-(R-D-M)}{N}. \tag{3.11}$$

Determining the multiprogramming conflicts involves measuring X_V and $\bar{X}_I$ from the reference stream. The measurement can be done by extending the recurrence/conflict single-pass technique. The miss ratio is then calculated by first calculating $M[C,B,S,q]$ using Equation (3.10) for a value of f_{CS}, then using the result to complete Equation (3.11).

3.3.3 Multiprogramming extensions to LRU stack operation

The extensions required to the recurrence/conflict single-pass technique that measure X_V and $\bar{X}_I$ are shown in Figure 3.23. The procedure for determining $X_V[C,B,S]$ is illustrated in Figure 3.22. The procedure operates as follows: When α is processed, if it is not a recurring reference (i.e., the test of Step 1 of Figure 3.23 fails), then it cannot be a victim since it cannot produce a hit. However, if α is a voluntary context switch

event, it is marked as such when it is pushed on the stack in Step 2 (marked references are shown using asterisks in Figure 3.22). If α is a recurring reference, $X_V[C, B, S]$ is

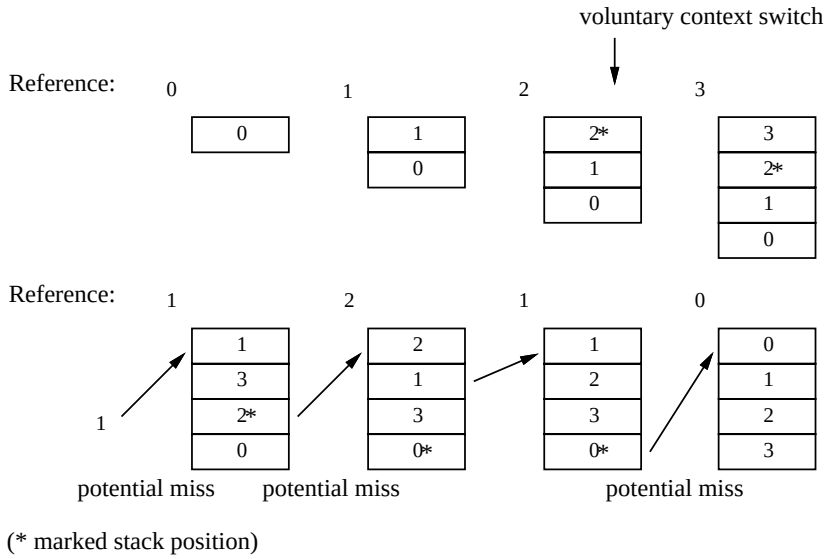

Figure 3.22: An example for voluntary context switch of the modified LRU stack operation.

conditionally incremented if a marked reference is encountered when the dimensional conflicts are calculated. $X_V[C, B, S]$ is only incremented for all dimensions in which α does not have a dimensional conflict. If $X_V[C, B, S]$ were incremented for all dimensions, a reference might be counted more than once as a conflict, once as a multiprogramming conflict and once as a dimensional conflict. Notice that the references immediately below marked references inherit the marking in the figure. This is done to insure that all subsequent recurring references that cross the context switch event are subject to a voluntary context switch. To maintain correct detection of voluntary context switches, the reference immediately below α on the stack is marked on repush if α was marked (Step 1.6 and its substeps of Figure 3.23).

The procedure for determining $\bar{X}_I[C, B, S, q]$ using an LRU stack is somewhat more complicated than that for determining $X_V[C, B, S]$. Recall that an involuntary context switch may occur between every reference. Let L, the *context switch distance*, be the number of potential involuntary context switch events for the recurring reference α at reference count k (i.e., $\alpha = w_B(k - L) = w_B(k)$). Let p_L be the probability that at least one involuntary context switch occurs between times $k - L$ and k. Then,

$$p_L = \sum_{j=1}^{L} \binom{L}{j} q^j (1-q)^{L-j}. \tag{3.12}$$

Define $n_L[C, B, S]$ to be the number of recurrences not subject to dimensional conflicts that have a context switch distance of L. Therefore,

$$\bar{X}_I[C, B, S, q] = E\Big[n_L[C, B, S]\Big] = \sum_L p_L n_L[C, B, S]. \tag{3.13}$$

Equation (3.13) expresses the expected number of potential victims due to involuntary context switching. The equation fits naturally into a stack-based method. The new metric $n_L[C, B, S]$ can be recorded by annotating the references on the stack. Figure 3.24 shows an example of calculating $\bar{X}_I[C, B, S]$. The figure shows that a counter of the number of context switch events affecting α is kept, defined as $c_I(\alpha)$. Initially and after a recurring reference is repushed, $c_I(\alpha) \leftarrow 1$ (Step 2.1 and 1.8 of Figure 3.23). In Step 1.3 and its substeps and Step 1.4, L is computed from one plus the sum of the counters of entries above α on the stack. (Notice that $c_I(\alpha)$ is not part of the calculation of L; Figure 3.24 illustrates this). In Step 1.5 and its substeps, $n_L[C, B, S]$ is incremented for all caches in which there are no dimensional conflicts. Let $S_B(k-1)[\Delta - 1] = \beta_0$, the address that is

```
1.      if α ∈ S_B(k − 1) then
1.1         vol_cs ← false
1.2         L ← 1
1.3         for i ← 0 to Δ do
1.3.1           β_i ← S_B(k − 1)[i]
1.3.2           if β_i marked as a voluntary context switch event then
1.3.2.1             vol_cs ← true
1.3.3           L ← L + c_I(β_i)
1.4         L ← L + 1
1.5         for all (C, B, S) without a dimensional conflict do
1.5.1           n_L[C, B, S] ← n_L[C, B, S] + 1
1.5.2           if vol_cs then
1.5.2.1             X_V[C, B, S] ← X_V[C, B, S] + 1
1.6         if α marked as a voluntary context switch event then
1.6.1           mark β_{Δ+1}
1.6.2           unmark α
1.7         c_I(β_{Δ−1}) ← c_I(β_{Δ−1}) + c_I(α)
1.8         c_I(α) ← 1
1.9         S_B(k) ← repush(S_B(k − 1), α),
2       else
2.1         c_I(α) ← 1
2.2         S_B(k) ← push(S_B(k − 1), α)
```

Figure 3.23: An LRU stack method modified for context switching.

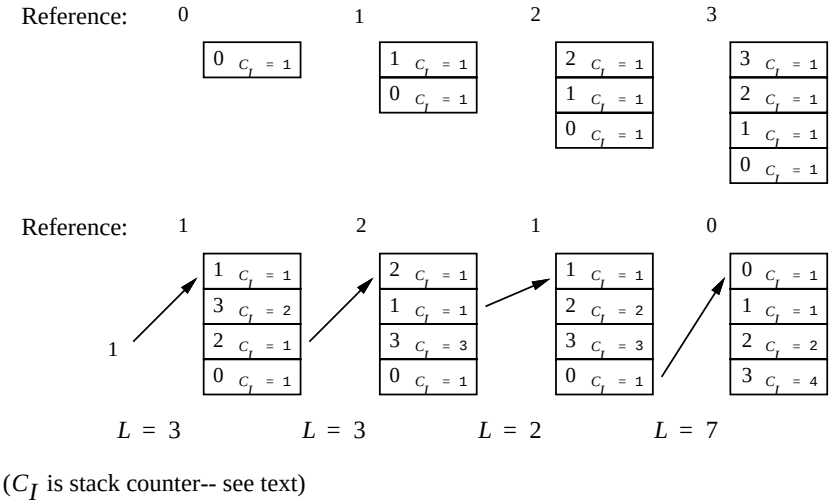

Figure 3.24: An example for involuntary context switching of the modified LRU stack operation.

directly above α in the stack $S_B(k-1)$. As a bookkeeping step, $c_I(\beta_0)$ is incremented by $c_I(\alpha)$ (Step 1.7). In this way, all of the references deeper in the stack than α in $S_B(k-1)$ will arrive at the correct context switch distance.

The algorithm shows $n_L[C,B,S]$ being maintained for all values of L. Not all values of L must be recorded using $n_L[C,B,S]$. Rather, power-of-two sized categories can be retained. The scheme used for the simulations that is presented below uses 14 categories. The first category contains $n_L[C,B,S]$ for $1 \leq L < 4$, following this, the ith category contains $n_L[C,B,S]$ for $2^{(i+2)} \leq L < 2^{i+3}$. This quantization scheme is based on observations of the distribution of $n_L[C,B,S]$ vs. L. The scheme does, however, produce error for small q, and this is commented on in the following section.

Notice that the calculation of $n_L[C,B,S]$ is independent of the context switching intensity distribution assumptions. The function used to calculate p_L in Equation (3.13)

need not be Equation (3.12). It is possible to substitute other context switching intensity distributions into Equation (3.13) without altering the presented single-pass method. The impact of this observation is that the method is more general than the assumption of uniformly distributed involuntary context switching of Equation (3.12).

The algorithm of Figure 3.23 is compatible with the no-state-loss sampling approach since the stack annotations (i.e., marked stack entries and maintenance of the c_I counters) may be included as part of the stack maintenance and need not be part of the `do_recurrence`() procedure. In this extension to sampling, the calculation of $n_L[C, B, S]$ and $X_V[C, B, S]$ would only occur inside samples, as was the calculation of $D'[C, B, S]$ for the no-state-loss approach. However, the context switching algorithm is not compatible with fill-flush or other traditional sampling approaches since the stack is not maintained between samples. This is an additional reason for the use of the no-state-loss approach instead of the fill-flush approach.

3.4 Empirical Results of Program Susceptibility

The validity of the single-pass method of the previous section is discussed below by comparing the method's results with those results from other techniques that have similar assumptions. The results from the model are presented and discussed for members of the SPEC89 benchmark set. The dimensional conflicts that occur due to different cache sizes are discussed in Section 3.4.4 to compare their performance degradation with that of context switching. Empirically observed values of the parameter f_{CS} are also

presented. It is found that $f_{CS} < 1$ for moderate multiprogramming loads, confirming the observation that $f_{CS} = 1$ produces overly pessimistic results.

3.4.1 The validity of the single-pass method

It is important to question whether the single-pass method extended to measure context switching produces performance estimates that are consistent with the assumptions made in Section 3.3.1. (Whether these assumptions are valid themselves is beyond the scope of this study.) The approach used in testing the validity of the method is to compare its predictions with methods used for traditional cache simulators.

One commonly used simulation technique to measure the effects of context switching is to flush the state of the simulation at context switching events [13],[14],[17]. It is clear that in this case, $f_{CS} = 1$ is assumed. The decision of when to flush the stack for voluntary context switch events is known since these are present in the trace. The decision of when to flush the cache for involuntary context switch events is done by distributing involuntary context switch events throughout the trace uniformly. This *random-interval simulation* flushes the contents of the stack based on a uniformly distributed random number with mean q. Note that the random-interval simulation requires a simulation for each value of q. The single-pass method does not have this restriction since it measures the effects of all q in one pass over the trace.

The random-interval simulation method approximates the assumptions of Section 3.3.1, except that the simulation produces results for one particular random distribution of context switch events across the trace. The single-pass method measures the average effect of all distributions. This discrepancy can be eased by averaging the results of several random-interval simulations, which are performed iteratively until the results converged.

Figure 3.25 present the difference between the random-interval simulation and the single-pass method for espresso, gcc and xlisp, expressed as the absolute error of the miss ratio, for $q = 0.01$ and $q = 0.001$, respectively. Only the absolute errors for fully associative caches are shown in the figure for brevity. Smaller levels of associativity were found to have lower error. The figure demonstrates that the difference between

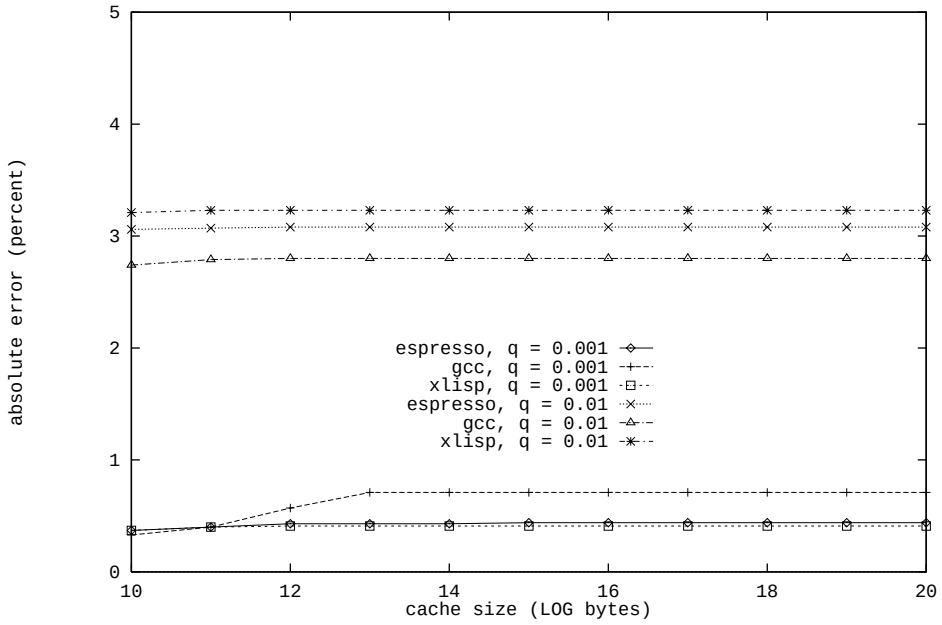

Figure 3.25: Absolute error of the miss ratio for random-interval simulation vs. single-pass method, $q = 0.01$ and $q = 0.001$.

the simulation and the model is approximately 3% to 3.5% for $q = 0.01$ and less than 1% for $q = 0.001$. The increase in error for smaller q values is due to the quantization of L discussed in the previous section. This error of 3% to 3.5% is large for $q = 0.01$; however, empirical evidence suggests that q values are typically in the range of $q = 0.001$ to $q = 0.0002$ (calculated from [36] and [37] and assuming one references made every five instructions). The single-pass technique using the power-of-two quantization of L produces results within 1% of the random-interval simulation for practical q values. This evidence suggests that the single-pass method produces results that are consistent with its assumptions.

3.4.2 Involuntary context switching susceptibility

It is useful to define $\Delta\rho = M[C, B, S, q]/N$ as a measure of benchmark susceptibility to context switching. This is the difference between the uniprogramming and multiprogramming miss ratios. Figure 3.26 presents $\Delta\rho$ for gcc and xlisp for block size 16 bytes. The results in the figure are for the cache $(31, 4, \infty)$. This large cache size is used to eliminate the effects of dimensional conflicts. Section 3.4.4 discusses the effects of dimensional conflicts. The figure considers only involuntary context switching. The effects of voluntary context switching are discussed in Section 3.4.3. Also, complete cache flushing ($f_{CS} = 1$) is used to emphasize the worst-case behavior (Section 3.4.5 discusses other values of f_{CS}). The figure demonstrates that when the intensity of context switching, q, is small, $\Delta\rho$ approaches zero such that context switching has little effect for $q \leq 0.0001$.

This value of q corresponds to an average context switching interval of 10,000 references. The gcc benchmark is slightly more susceptible to context switching than the xlisp benchmark for $q = 0.1$. This situation reverses itself and xlisp becomes more susceptible for $q > 0.01$. This phenomenon can be explained with the cumulative distribution of n_L vs. L, which is plotted in Figure 3.27. The figure has a logarithmic axis for the independent variable, L. From the figure it is apparent that xlisp has a higher number of recurrences for $L \leq 2^5$: 85% for gcc vs. 87.4% for xlisp. This would imply that a context switch frequency of greater than every $2^5 = 32$ references would impact xlisp more than gcc. This explains the behavior observed in Figure 3.26.

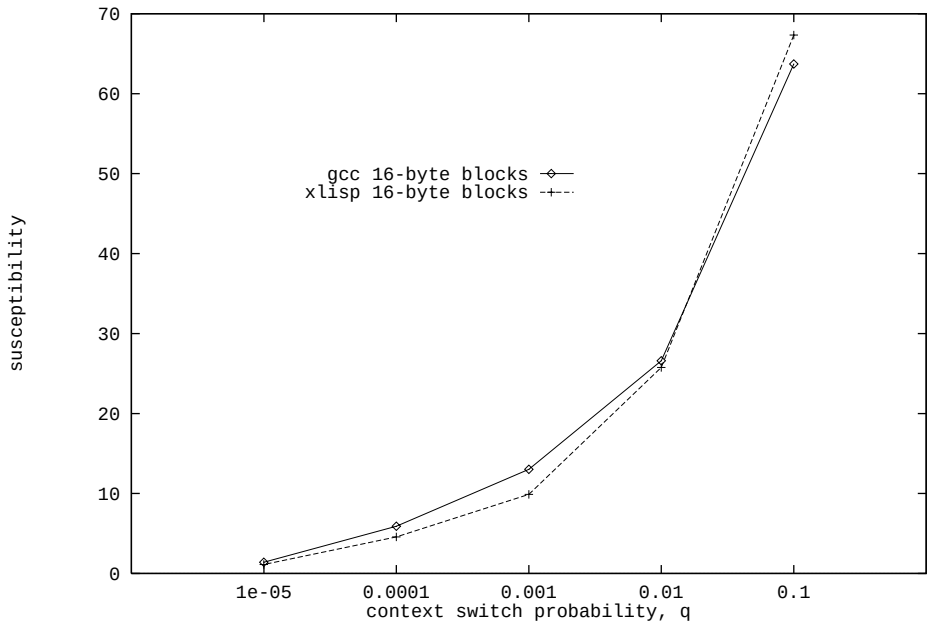

Figure 3.26: $\Delta\rho$ (involuntary) of gcc and xlisp vs. q for block size 16 bytes.

Figures 3.28 and 3.29 present $\Delta\rho$ for gcc, espresso and xlisp for block sizes 32 bytes and 64 bytes, respectively. The data for all the benchmarks are presented in Table 3.1. The

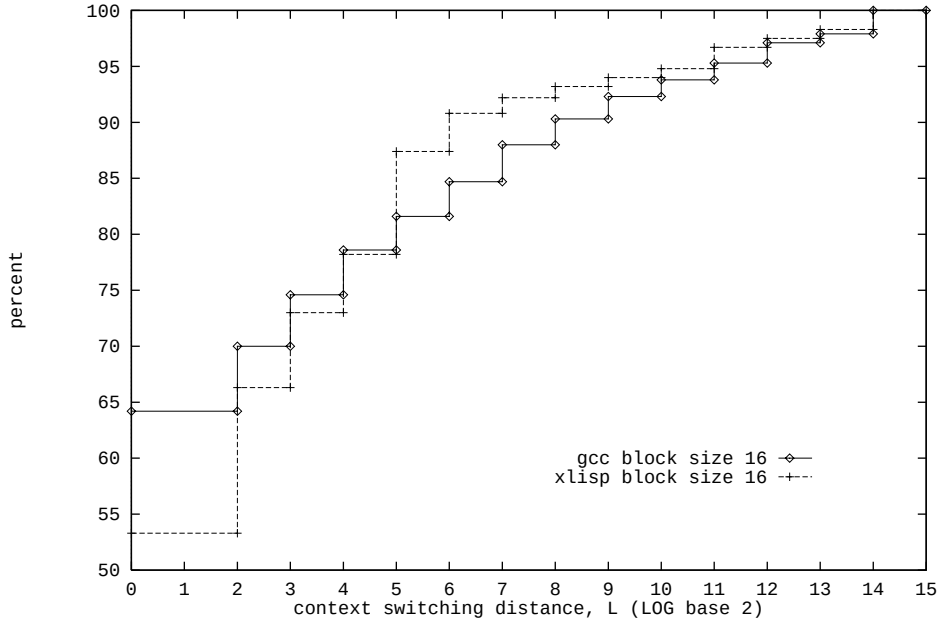

Figure 3.27: Cumulative distribution of n_L vs. L for block size 16 bytes.

Table 3.1: Involuntary context switching susceptibility ($\Delta\rho$) for caches $(31, -, \infty)$.

	$\Delta\rho$								
	$B = 4$			$B = 5$			$B = 6$		
Benchmark	10^{-2}	10^{-3}	10^{-4}	10^{-2}	10^{-3}	10^{-4}	10^{-2}	10^{-3}	10^{-4}
doduc	37.8%	16.1%	5.34%	28.9%	10.9%	3.31%	23.9%	8.03%	2.22%
eqntott	5.98%	2.41%	1.27%	5.21%	1.98%	1.04%	4.60%	1.57%	0.79%
espresso	23.0%	11.2%	4.88%	17.5%	7.23%	2.94%	14.04%	4.71%	1.71%
gcc	26.6%	13.0%	5.91%	22.2%	9.52%	3.85%	19.1%	7.22%	2.59%
matrix300	7.78%	5.76%	3.54%	4.91%	3.01%	1.79%	3.46%	1.63%	0.91%
xlisp	25.8%	9.89%	4.56%	21.3%	7.21%	2.91%	18.0%	5.39%	1.89%
Average	21.2%	9.73%	4.35%	16.7%	6.64%	2.64%	13.9%	4.76%	1.69%
Std. dev.	12.2%	4.96%	1.66%	9.72%	3.52%	1.04%	8.24%	2.72%	0.71%

corresponding cumulative distribution functions of n_L vs. L are presented in Figures 3.30 and 3.31, respectively. Together, Figures 3.26–3.29 demonstrate that the susceptibility to context switching decreases as block size increases. From Table 3.1, for $q = 0.01$, the difference in the miss ratio is 21.2% for block size 16 bytes and 13.9% for block size 32 bytes, on average. One possible reason for this is that the block reference streams for larger block sizes have smaller context switching distances. This occurs since more references occupy the same cache block for larger block sizes than for smaller block sizes.

Comparison between the benchmarks reveals significant variance in susceptibility. The matrix300 and eqntott benchmarks have the lowest change in the miss ratio, whereas benchmarks such as doduc and xlisp are quite sensitive to the value of q. For $q = 0.01$, $\Delta\rho = 37.8\%$ for doduc compared to 5.98% for xlisp. This confirms that susceptibility is a characteristic of the benchmark and that workload choice influences the observed effects of context switching.

3.4.3 Voluntary context switching susceptibility

The susceptibility of the benchmarks to voluntary context switching effects is relatively small compared to the involuntary effects. This can be seen in Table 3.2, which presents the voluntary susceptibility ($\Delta\rho$) for fully associative caches of the largest dimension for gcc and espresso. The largest-dimensional fully associative caches were selected so that $\Delta\rho$ would be at its maximum since no dimensional conflicts occur. Table 2.3 in Chapter 2 demonstrates that the occurrences of voluntary context switches are rare

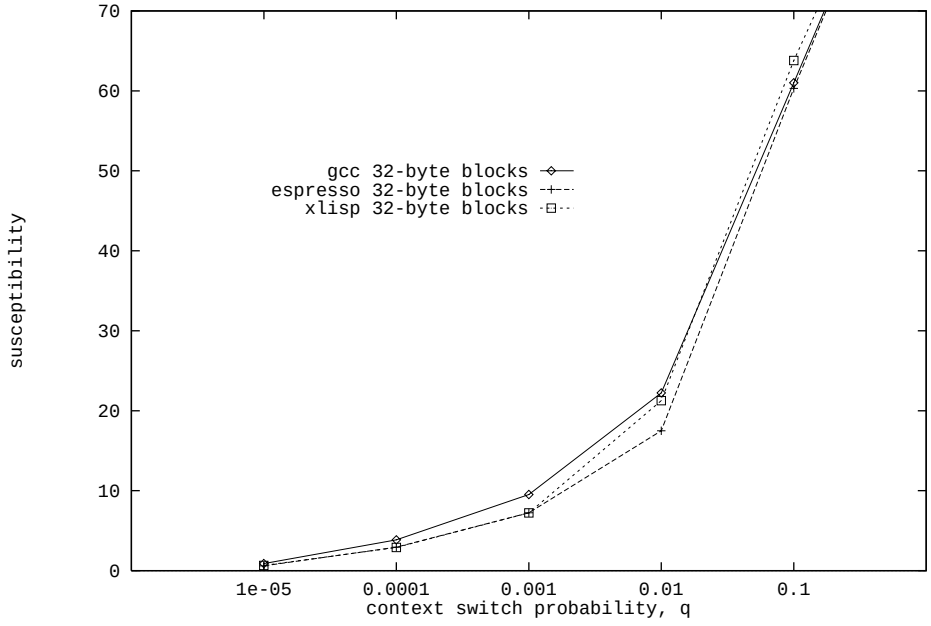

Figure 3.28: $\Delta\rho$ (involuntary) of gcc, espresso and xlisp vs. q for block size 32 bytes.

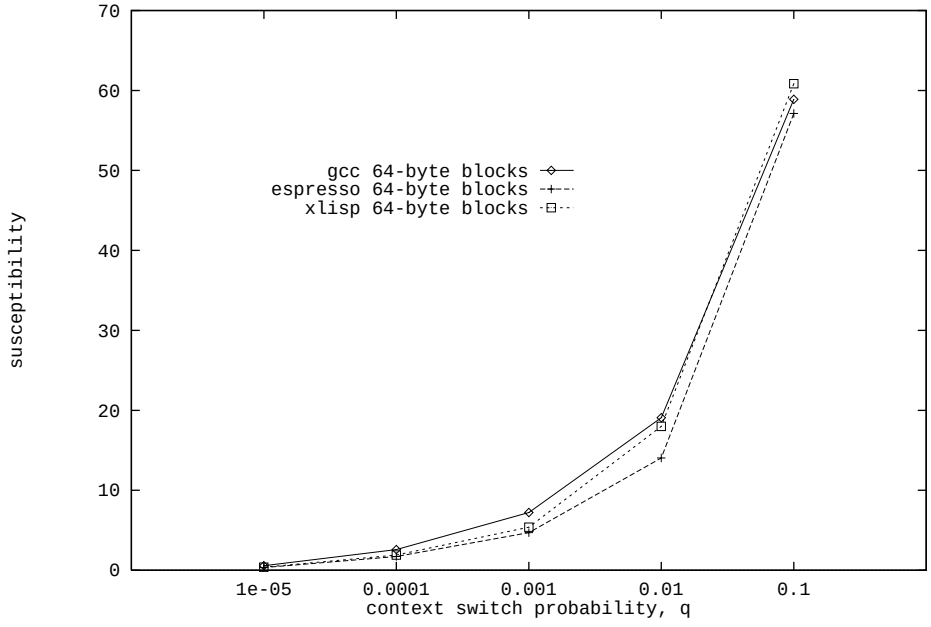

Figure 3.29: $\Delta\rho$ (involuntary) of gcc, espresso and xlisp vs. q for block size 64 bytes.

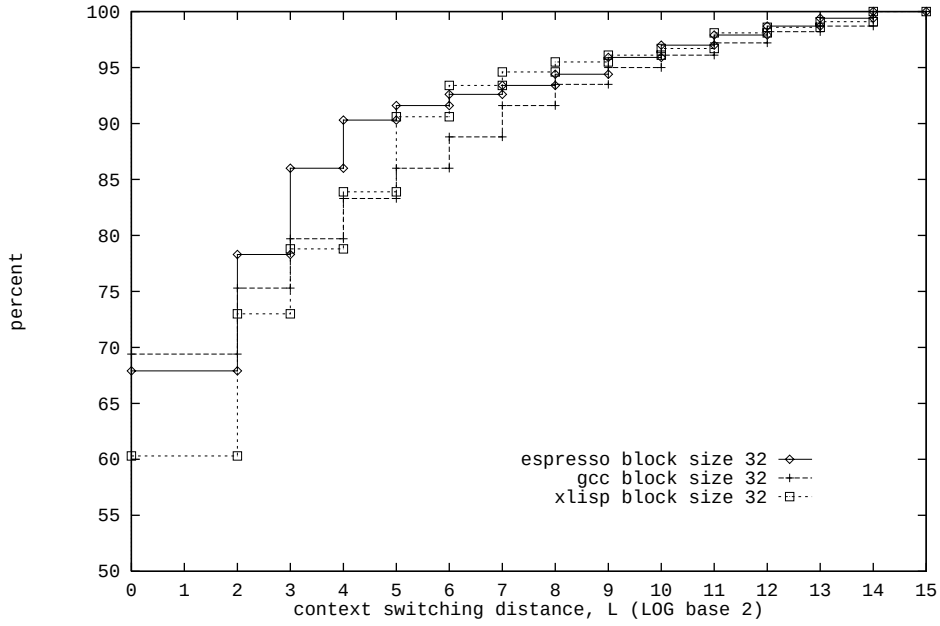

Figure 3.30: Cumulative distribution of n_L vs. L for block size 32 bytes.

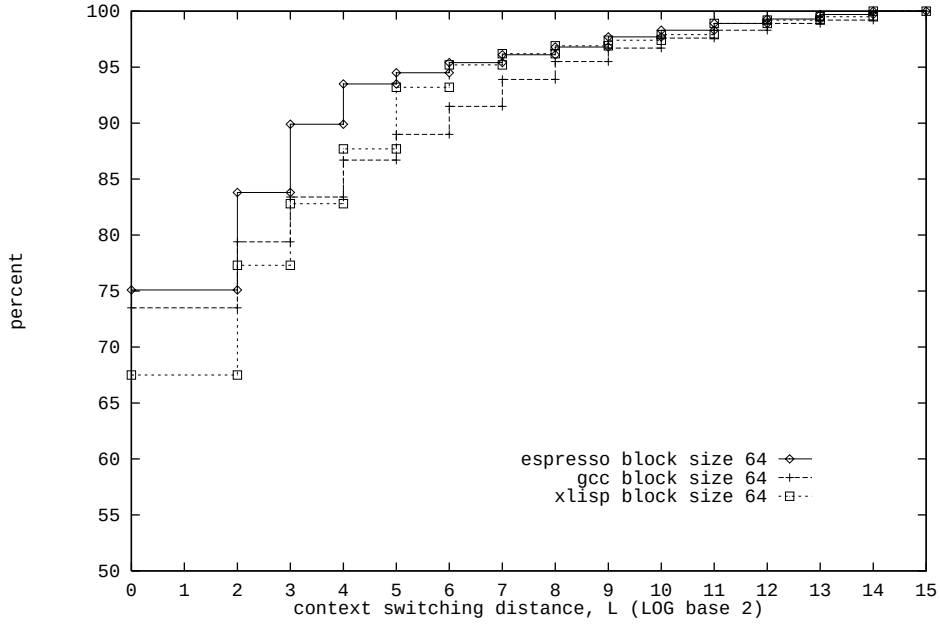

Figure 3.31: Cumulative distribution of n_L vs. L for block size 64 bytes.

Table 3.2: Voluntary context switching susceptibility vs. block size.

	Block size (bytes)		
Benchmark	16	32	64
doduc	0.07%	0.04%	0.03%
espresso	0.03%	0.02%	0.01%
eqntott	5.6×10^{-3}%	3.8×10^{-3}%	2.4×10^{-3}%
gcc	3.1%	2.0%	1.4%
matrix300	1.9×10^{-3}%	1.8×10^{-3}%	1.7×10^{-3}%
xlisp	6.0×10^{-4}%	6.0×10^{-4}%	6.0×10^{-4}%

for these benchmarks as well as for other members of the SPEC89 set. This explains the small susceptibility due to voluntary context switches. This may well be an artifact of benchmark selection and should not be taken as a general statement that voluntary context switches do not have much effect.

One of the benchmarks, gcc, is selected to serve as an example for the discussions that follow to illustrate the behavior of the susceptibility model.

3.4.4 Dimensional conflict effects

The dimensional conflicts have been excluded from consideration thus far by considering large, fully associative caches to isolate the effects of context switching. The relative importance of dimensional conflicts to multiprogramming conflicts is interesting because some cache designs may be more resilient to context switching than others due to the influences of dimensional conflicts. Consider caches of size 1K bytes: it is selected as a worst case since it should experience a high percentage of dimensional conflicts due to its extremely small size. Figure 3.32 shows $\Delta\rho$ vs. q for gcc using caches of 1K-bytes and

several associativities. Calculating the miss ratios for the uniprogrammed case for gcc

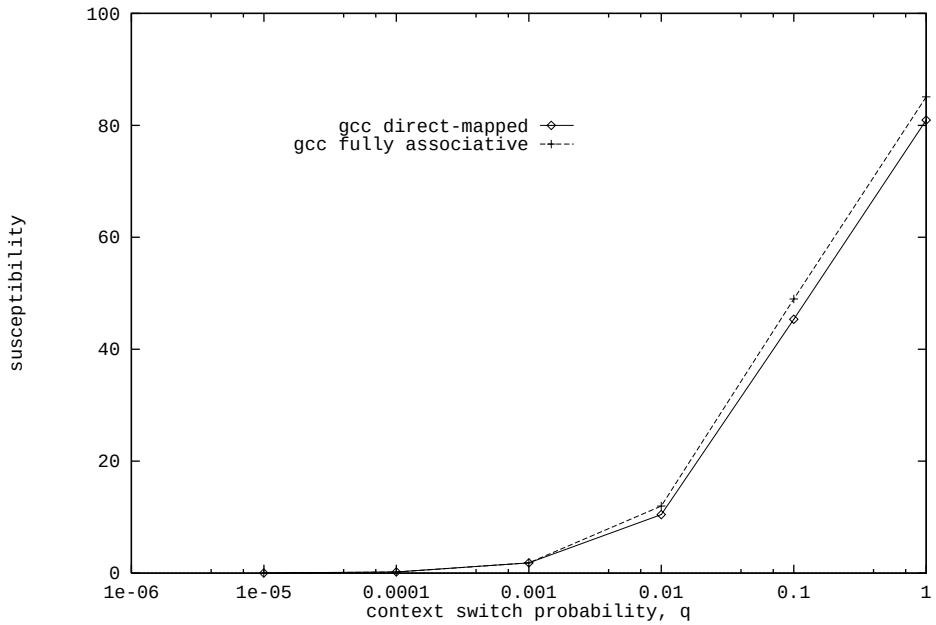

Figure 3.32: $\Delta\rho$ (involuntary) of gcc for caches $(10, 4, -)$.

reveals a variation of 18% for $(10, 4, 0)$ to 15% for $(10, 4, \infty)$ (these data are not shown in the figure). However, there is much less variation in $\Delta\rho$ apparent in Figure 3.32. This same effect is apparent from the data collected for the other benchmarks.

The above data suggest that dimensional conflicts dominate over context switching effects for small caches. To quantify this, the ratio of the multiprogramming conflicts to the dimensional conflicts, $M[C, B, S, q]/D[C, B, S]$, can be used as a measure of the relative impact of multiprogramming conflicts to dimensional conflicts. This ratio is plotted against q using caches of dimension $(10, 4, -)$ and $(13, 4, -)$ for gcc and the results are shown in Figure 3.33. The figure demonstrates that for small q, dimensional conflicts dominate. The two kinds of conflicts have equal effect (i.e., $M/D = 1.0$) for $q \approx 0.02$ with

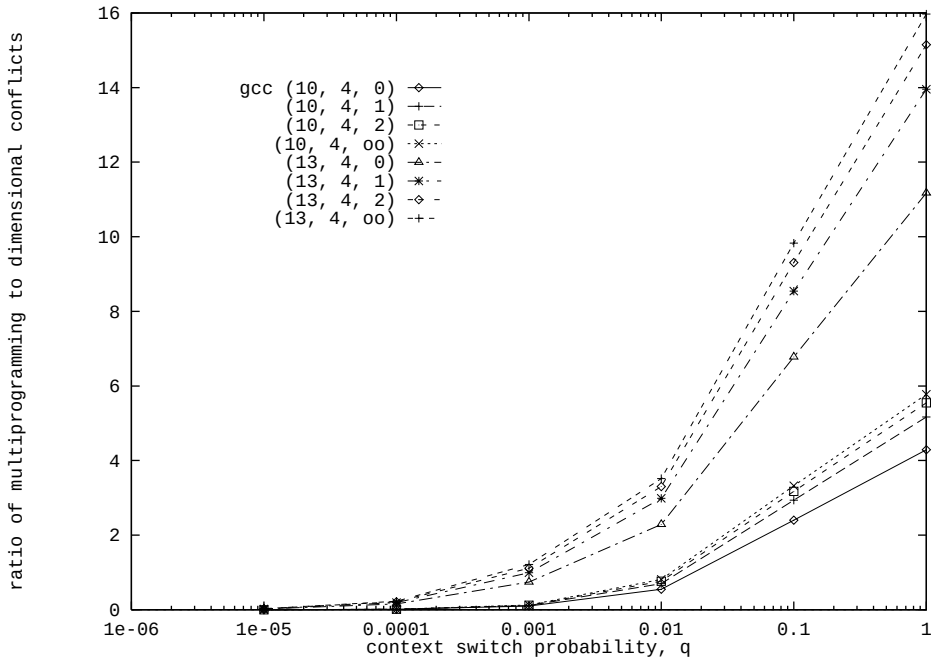

Figure 3.33: M/D vs. q for gcc, caches $(10, 4, -)$ and $(13, 4, -)$.

caches $(10, 4, -)$ and for $q \approx 0.00003$ with caches $(13, 4, -)$. As associativity increases, the performance depends more on the multiprogramming conflicts than on dimensional conflicts. Also, the importance of associativity increases with overall cache size. This implies that when associativity is used, multiprogramming effects can decide the cache size, which is similar to the observations of [19] concerning associativity.

To show that the effects observed are not an artifact of the test cache sizes of 1K and 8K bytes, Figure 3.34 presents M/D ratios for various cache and block sizes. Any value of q would have been sufficient to demonstrate the general relationship between M/D and C. The data from Figure 3.33 were used to select $q = 0.02$ for Figure 3.34. Since in this region the effects of associativity are relatively minor, the associativity is fixed at two-way associative (e.g., all caches $(-, -, 1)$). (Note that here, unlike the earlier

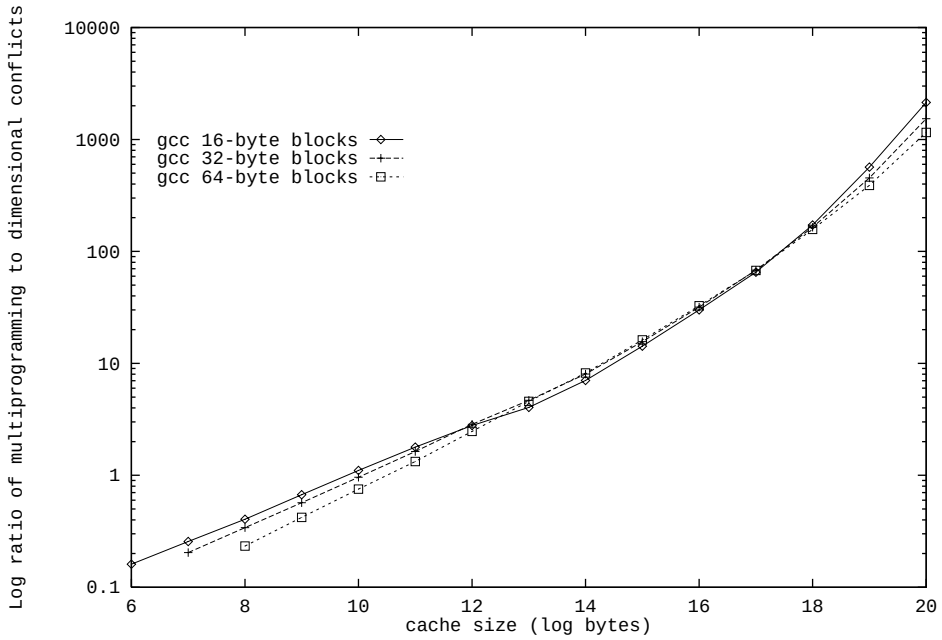

Figure 3.34: $\log(M/D)$ vs. cache size, for various block sizes ($q = 0.02$).

figure, M/D is presented using a logarithmic scale.) From the figure, it is immediately apparent that the worst-case relative impact of multiprogramming (i.e., M/D) increases approximately linearly with cache size (both axes are logarithmic). Also, as a refinement of the observations made in Section 3.4.2, block size is inversely proportional to program susceptibility for small caches (less than 2K bytes). However, block size appears to be directly proportional to program susceptibility for moderately large cache sizes (4K bytes up to 256K bytes), after which the trend reverses itself again.

3.4.5 Measurement of fraction of cache flush (f_{CS})

This section has assumed $f_{CS} = 1$ in order to measure the worst-case susceptibility of programs to context switching. This section presents some empirical estimates of

the f_{CS} parameter. The method used simulates a multiprogrammed system using a trace composed of interleaved sections of traces taken from several benchmarks. At each reference, the interleaver makes a decision whether to continue processing the current trace or to switch to another waiting trace. This probability is assumed to be uniformly distributed with mean q.

The values of f_{CS} for each benchmark can be derived by comparing the number of misses for a uniprogramming cache to the observed number of misses for the multiprogramming cache. Any additional misses in the multiprocessor case must be due to context switching. Let $D_u[C, B, S]$ represent the dimensional misses from the uniprogramming trace and $D_m[C, B, S]$ represent the dimensional misses measured from the multiprogramming trace. Let $\hat{M}$ represent the estimated multiprogramming conflicts. Then,

$$\hat{M}[C, B, S, q] = D_m[C, B, S] - D_u[C, B, S]. \tag{3.14}$$

Note that it can always be assumed that $D_m[C, B, S] > D_u[C, B, S]$ since any dimensional conflicts for a benchmark trace must come as a result of the interleaving of traced events. Using Equation (3.10) and neglecting the X_V term as justified by the experimental results of Section 3.4.3, then,

$$\hat{f}_{CS} = \frac{D_m[C, B, S] - D_u[C, B, S]}{X_I[C, B, S, q]}, \tag{3.15}$$

where $\hat{f}_{CS}$ is the experimental value for f_{CS}. Equation (3.15) requires knowledge of X_I, which would require use of the algorithm of Figure 3.24.

The two experiments interleave espresso, gcc, and xlisp with $q = 0.01$ and $q = 0.001$. A block size of 32 bytes was assumed for these experiments. Values of f_{CS} across cache size and associativities for the espresso benchmark are presented in Figures 3.35 ($q = 0.01$) and 3.36 ($q = 0.001$) from the perspective of the espresso benchmark. It is clear from

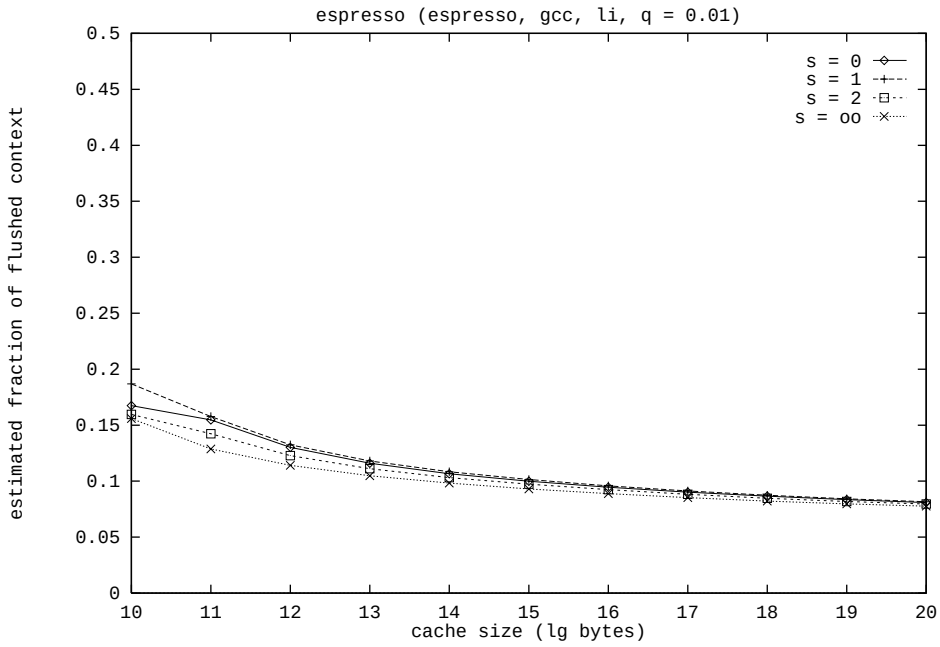

Figure 3.35: f_{CS} vs. cache size for espresso, $q = 0.01$.

these two figures that f_{CS} is a function of cache size, as suggested by Equation (3.15). For small caches, $f_{CS} \approx 13\%$–18%, whereas for large caches, $f_{CS} \approx 8\%$, when $q = 0.01$. The effects of associativity on f_{CS} are less pronounced than the effects of cache size for $q = 0.01$ (Figure 3.35). For $q = 0.001$ (Figure 3.36), $f_{CS} \approx 0$ large, fully associative cache sizes. This is not true for smaller associativities, since dimensional conflicts occur between the references of the three benchmarks regardless of cache size. The effects of associativity also become less noticeable as cache size increases, possibly because fewer

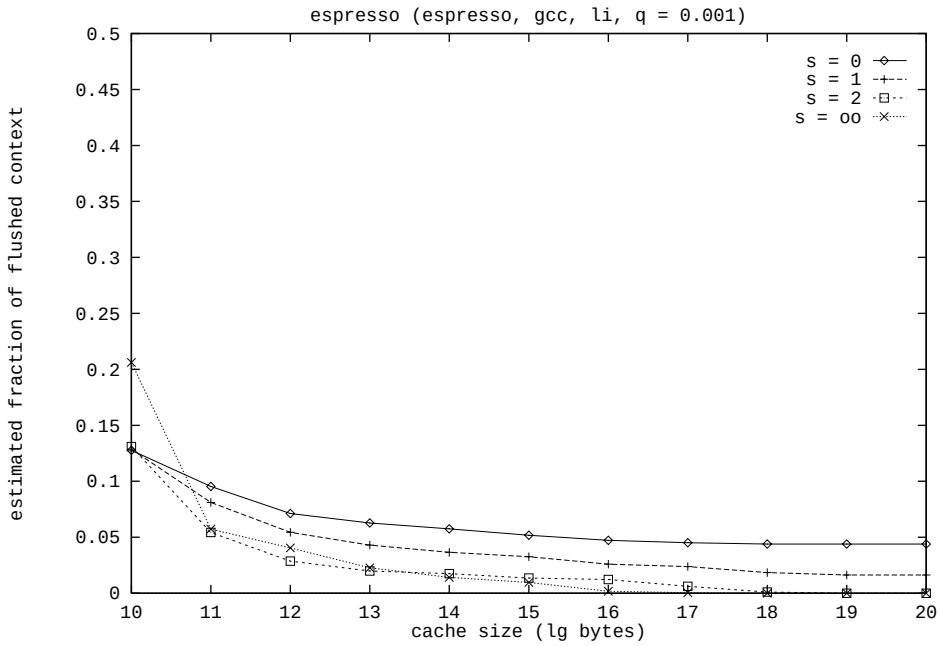

Figure 3.36: f_{CS} vs. cache size for espresso, $q = 0.001$.

dimensional conflicts occur between the references from espresso and those of gcc and xlisp.

These experiments show that $f_{CS} = 1$ is a pessimistic assumption under moderate load. When workloads are decomposed into workload elements or workload elements are taken from standard benchmark sets such as SPEC89, it is not possible to predict the different combinations of benchmarks that may execute together in the final system. In this situation, a conservative assumption such as $f_{CS} = 1$ is appropriate. The results for the susceptibility measures presented above for $f_{CS} = 1$ suggest that the difference in the miss ratio will not change considerably for values of $q \leq 10^4$. If designs are selected to satisfy a required maximum miss ratio, this observation suggests that selecting prototypes

with $f_{CS} = 1$ adds degree of tolerance to context switching to the designs with small increase in cost. This cost is verified in the following section.

3.5 Examples of Memory System Prototyping

This section presents several prototypes of the first-level data cache for a hypothetical memory system. The memory system evaluation techniques of this chapter are used to derive the miss ratios of a space of potential cache dimensions in the range of single-block caches to 2 gigabytes with block sizes of 16, 32, 64 bytes. Associativities of one-way, two-way, four-way and fully associative are explored. For each benchmark, this space constitutes 324 possible cache designs (after removing invalid cache dimensions).

The tool used to collect the traces for these results is not capable of generating traces of the instruction referencing patterns that are independent of instruction-set encodings (the instructions themselves are, however, as described in Chapter 2). Rather than present instruction cache designs that are specific to a particular architecture, this section concentrates on data cache designs. The methods used to prototype the caches are independent of the sources of the trace, so that the use of data address traces here does not limit the applicability of the techniques of this chapter.

3.5.1 Data cache prototypes

Data caches are selected based on a performance criterion. This performance criterion is expressed by placing an upper limit on the miss ratio, ρ, and examining the cache sizes

for each set size and block size. A cache (C, B, S) is selected such that C is the largest C that satisfies $\rho(C, B, S) \leq \hat{\rho}$. The parameter, $\hat{\rho}$, is the *miss ratio selection criterion.* Tables 3.3, 3.4 and 3.5 present the prototype data cache designs for $\hat{\rho} = 0.10$, 0.05 and 0.01, respectively. In some situations the design criterion cannot be met by any cache size since the miss ratio for an infinitely sized cache for the benchmark exceeds the value of $\hat{\rho}$. These situations have been marked with an asterisk in the tables.

Several observations can be derived from these designs:

- For a criterion of $\hat{\rho} = 0.10$ (Table 3.3), the required size for direct-mapped caches is often two to four times the size of the fully associative size. This generalization is similarly evident for $\hat{\rho} = 0.05$ and $\hat{\rho} = 0.01$ (Tables 3.4 and 3.5).

- For $\hat{\rho} = 0.10$ and $\hat{\rho} = 0.05$, increased block size from 16 bytes to 32 bytes often halves the required cache size.

- The maximum cache sizes required are 16KB for $\hat{\rho} = 0.10$ (Table 3.3), 32KB for $\hat{\rho} = 0.05$ (Table 3.4) and 256MB for $\hat{\rho} = 0.01$ (Table 3.5).

- If the cost of a memory system is measured in overall size in bytes, the design parameters of Table 3.4 (achieving close to 5% miss ratios) are much less expensive than the design parameters of Table 3.5 (achieving close to 1% miss ratios).

- The most-demanding benchmarks in terms of cache size are gcc and matrix300. (Note that this is for a simplistic layout of data for matrix300– the required cache size of matrix300 can be reduced using data partitioning techniques [38].)

Table 3.3: Prototype data cache dimensions for criterion $\hat{\rho} = 0.10$.

Benchmark	Prototype data cache dimensions			
doduc	(14, 4, 0)	(12, 4, 1)	(12, 4, 2)	(12, 4, ∞)
	(13, 5, 0)	(12, 5, 1)	(12, 5, 2)	(12, 5, ∞)
	(13, 6, 0)	(12, 6, 1)	(12, 6, 2)	(12, 6, ∞)
eqntott	(8, 4, 0)	(7, 4, 1)	(7, 4, 2)	(8, 4, ∞)
	(9, 5, 0)	(7, 5, 1)	(8, 5, 2)	(9, 5, ∞)
	(9, 6, 0)	(8, 6, 1)	(9, 6, 2)	(10, 6, ∞)
espresso	(12, 4, 0)	(11, 4, 1)	(11, 4, 2)	(11, 4, ∞)
	(11, 5, 0)	(10, 5, 1)	(10, 5, 2)	(9, 5, ∞)
	(11, 6, 0)	(10, 6, 1)	(10, 6, 2)	(10, 6, ∞)
gcc	(13, 4, 0)	(12, 4, 1)	(12, 4, 2)	(12, 4, ∞)
	(12, 5, 0)	(12, 5, 1)	(12, 5, 2)	(11, 5, ∞)
	(13, 6, 0)	(12, 6, 1)	(12, 6, 2)	(11, 6, ∞)
matrix300	(14, 4, 0)	(13, 4, 1)	(13, 4, 2)	(13, 4, ∞)
	(9, 5, 0)	(8, 5, 1)	(8, 5, 2)	(9, 5, ∞)
	(9, 6, 0)	(8, 6, 1)	(9, 6, 2)	(10, 6, ∞)
xlisp	(12, 4, 0)	(11, 4, 1)	(10, 4, 2)	(10, 4, ∞)
	(11, 5, 0)	(11, 5, 1)	(10, 5, 2)	(10, 5, ∞)
	(12, 6, 0)	(11, 6, 1)	(11, 6, 2)	(11, 6, ∞)

Table 3.4: Prototype data cache dimensions for criterion $\hat{\rho} = 0.05$.

Benchmark	Prototype data cache dimensions			
doduc	(15, 4, 0)	(13, 4, 1)	(13, 4, 2)	(13, 4, ∞)
	(14, 5, 0)	(13, 5, 1)	(13, 5, 2)	(13, 5, ∞)
	(14, 6, 0)	(13, 6, 1)	(13, 6, 2)	(13, 6, ∞)
eqntott	(10, 4, 0)	(9, 4, 1)	(8, 4, 2)	(8, 4, ∞)
	(10, 5, 0)	(9, 5, 1)	(9, 5, 2)	(9, 5, ∞)
	(11, 6, 0)	(10, 6, 1)	(9, 6, 2)	(10, 6, ∞)
espresso	(14, 4, 0)	(14, 4, 1)	(14, 4, 2)	(14, 4, ∞)
	(14, 5, 0)	(13, 5, 1)	(12, 5, 2)	(12, 5, ∞)
	(13, 6, 0)	(12, 6, 1)	(11, 6, 2)	(11, 6, ∞)
gcc	(15, 4, 0)	(14, 4, 1)	(14, 4, 2)	(14, 4, ∞)
	(14, 5, 0)	(14, 5, 1)	(13, 5, 2)	(13, 5, ∞)
	(14, 6, 0)	(13, 6, 1)	(13, 6, 2)	(13, 6, ∞)
matrix300	(*, 4, 0)	(*, 4, 1)	(*, 4, 2)	(*, 4, ∞)
	(14, 5, 0)	(13, 5, 1)	(13, 5, 2)	(13, 5, ∞)
	(11, 6, 0)	(9, 6, 1)	(9, 6, 2)	(10, 6, ∞)
xlisp	(13, 4, 0)	(13, 4, 1)	(13, 4, 2)	(13, 4, ∞)
	(13, 5, 0)	(12, 5, 1)	(12, 5, 2)	(11, 5, ∞)
	(13, 6, 0)	(12, 6, 1)	(12, 6, 2)	(11, 6, ∞)

Table 3.5: Prototype data cache dimensions for criterion $\hat{\rho} = 0.01$.

Benchmark	Prototype data cache dimensions			
doduc	(17, 4, 0)	(16, 4, 1)	(15, 4, 2)	(15, 4, ∞)
	(17, 5, 0)	(15, 5, 1)	(15, 5, 2)	(15, 5, ∞)
	(17, 6, 0)	(15, 6, 1)	(15, 6, 2)	(15, 6, ∞)
eqntott	(16, 4, 0)	(15, 4, 1)	(15, 4, 2)	(15, 4, ∞)
	(16, 5, 0)	(15, 5, 1)	(14, 5, 2)	(14, 5, ∞)
	(15, 6, 0)	(14, 6, 1)	(13, 6, 2)	(13, 6, ∞)
espresso	(16, 4, 0)	(15, 4, 1)	(15, 4, 2)	(15, 4, ∞)
	(16, 5, 0)	(15, 5, 1)	(15, 5, 2)	(15, 5, ∞)
	(16, 6, 0)	(15, 6, 1)	(15, 6, 2)	(15, 6, ∞)
gcc	(18, 4, 0)	(17, 4, 1)	(17, 4, 2)	(17, 4, ∞)
	(17, 5, 0)	(17, 5, 1)	(16, 5, 2)	(16, 5, ∞)
	(17, 6, 0)	(16, 6, 1)	(16, 6, 2)	(16, 6, ∞)
matrix3	(*, 4, 0)	(*, 4, 1)	(*, 4, 2)	(*, 4, ∞)
	(*, 5, 0)	(*, 5, 1)	(*, 5, 2)	(*, 5, ∞)
	(*, 6, 0)	(*, 6, 1)	(*, 6, 2)	(*, 6, ∞)
xlisp	(18, 4, 0)	(18, 4, 1)	(16, 4, 2)	(16, 4, ∞)
	(17, 5, 0)	(16, 5, 1)	(15, 5, 2)	(16, 5, ∞)
	(16, 6, 0)	(15, 6, 1)	(15, 6, 2)	(15, 6, ∞)

(* criterion could not be met)

3.5.2 Prototypes for multiprogramming environments

Tables 3.6 and 3.7 present data cache prototypes for the memory system selected using the criteria $\hat{\rho} = 0.10$ and $\hat{\rho} = 0.05$, respectively. Performance for these prototypes is simulated using the context switching single-pass method of the previous section. The parameters of the multiprogramming workload are a context switching intensity of $q = 0.0001$ and a flush percentage of $f_{CS} = 1.0$. Several observations can be made from these results:

- The following prototypes in Table 3.6 exceed sizes of the designs of Table 3.3:
  - (13, 4, 1) for doduc,
  - (13, 4, 0), (12, 4, 1), and (12,5,0) for espresso,
  - (12, 6, ∞) for gcc, and,
  - (12, 5, 0) for xlisp.

  all other prototypes are identical.

- Comparison of Table 3.7 and Table 3.4 produces the following observations:
  - 30 of the 72 multiprogramming designs exceed the requirements of Table 3.4.
  - Only the eqntott benchmark designs did not change from those of Table 3.4.
  - For three benchmarks, doduc, espresso, and matrix300, the criteria of $\hat{\rho} = 0.05$ cannot be met for block size $B = 4$ (16 bytes).

Table 3.6: Prototype data cache dimensions for criterion $\hat{\rho} = 0.10$, multiprogramming parameters: $q = 0.0001$ and $f_{CS} = 1.0$.

Benchmark	Prototype data cache dimensions			
doduc	(14, 4, 0)	(13, 4, 1)	(12, 4, 2)	(12, 4, ∞)
	(13, 5, 0)	(12, 5, 1)	(12, 5, 2)	(12, 5, ∞)
	(13, 6, 0)	(12, 6, 1)	(12, 6, 2)	(12, 6, ∞)
eqntott	(8, 4, 0)	(7, 4, 1)	(7, 4, 2)	(8, 4, ∞)
	(9, 5, 0)	(7, 5, 1)	(8, 5, 2)	(9, 5, ∞)
	(9, 6, 0)	(8, 6, 1)	(9, 6, 2)	(10, 6, ∞)
espresso	(13, 4, 0)	(12, 4, 1)	(11, 4, 2)	(11, 4, ∞)
	(12, 5, 0)	(10, 5, 1)	(10, 5, 2)	(9, 5, ∞)
	(11, 6, 0)	(10, 6, 1)	(10, 6, 2)	(10, 6, ∞)
gcc	(13, 4, 0)	(12, 4, 1)	(12, 4, 2)	(12, 4, ∞)
	(13, 5, 0)	(12, 5, 1)	(12, 5, 2)	(11, 5, ∞)
	(13, 6, 0)	(12, 6, 1)	(12, 6, 2)	(12, 6, ∞)
matrix300	(14, 4, 0)	(13, 4, 1)	(13, 4, 2)	(13, 4, ∞)
	(9, 5, 0)	(8, 5, 1)	(8, 5, 2)	(9, 5, ∞)
	(9, 6, 0)	(8, 6, 1)	(9, 6, 2)	(10, 6, ∞)
xlisp	(12, 4, 0)	(11, 4, 1)	(10, 4, 2)	(10, 4, ∞)
	(12, 5, 0)	(11, 5, 1)	(10, 5, 2)	(10, 5, ∞)
	(12, 6, 0)	(11, 6, 1)	(11, 6, 2)	(11, 6, ∞)

Table 3.7: Prototype data cache dimensions for criterion $\hat{\rho} = 0.05$, multiprogramming parameters: $q = 0.0001$ and $f_{CS} = 1.0$.

Benchmark	Prototype data cache dimensions			
doduc	(*, 4, 0)	(*, 4, 1)	(*, 4, 2)	(*, 4, ∞)
	(15, 5, 0)	(13, 5, 1)	(13, 5, 2)	(13, 5, ∞)
	(15, 6, 0)	(13, 6, 1)	(13, 6, 2)	(13, 6, ∞)
eqntott	(10, 4, 0)	(9, 4, 1)	(8, 4, 2)	(8, 4, ∞)
	(10, 5, 0)	(9, 5, 1)	(9, 5, 2)	(9, 5, ∞)
	(11, 6, 0)	(10, 6, 1)	(9, 6, 2)	(10, 6, ∞)
espresso	(18, 4, 0)	(16, 4, 1)	(15, 4, 2)	(15, 4, ∞)
	(15, 5, 0)	(13, 5, 1)	(13, 5, 2)	(13, 5, ∞)
	(13, 6, 0)	(12, 6, 1)	(11, 6, 2)	(11, 6, ∞)
gcc	(*, 4, 0)	(*, 4, 1)	(*, 4, 2)	(*, 4, ∞)
	(16, 5, 0)	(14, 5, 1)	(14, 5, 2)	(14, 5, ∞)
	(15, 6, 0)	(14, 6, 1)	(13, 6, 2)	(13, 6, ∞)
matrix300	(*, 4, 0)	(*, 4, 1)	(*, 4, 2)	(*, 4, ∞)
	(15, 5, 0)	(13, 5, 1)	(13, 5, 2)	(13, 5, ∞)
	(11, 6, 0)	(9, 6, 1)	(9, 6, 2)	(10, 6, ∞)
xlisp	(15, 4, 0)	(14, 4, 1)	(14, 4, 2)	(13, 4, ∞)
	(13, 5, 0)	(12, 5, 1)	(12, 5, 2)	(12, 5, ∞)
	(13, 6, 0)	(12, 6, 1)	(12, 6, 2)	(11, 6, ∞)

(* criterion could not be met)

3.6 Summary of Memory System Prototyping

The purpose of this chapter has been to present techniques to prototype cache-based memory system hierarchies. The recurrence/conflict version of a single-pass cache simulation method was introduced and extended to satisfy the needs of prototyping. The two major extensions are the adaptation of statistical sampling techniques from traditional cache simulation methods to single-pass methods, and an extension to measure the susceptibility of a benchmark to context switching.

Two techniques were presented in this chapter for statistical sampling of address traces using the single-pass methods. The *fill-flush* technique was adapted directly from traditional cache simulation techniques and its accuracy and speed were presented. A new technique, the *no-state-loss* technique, was developed. This technique exploits properties of the single-pass method to achieve very high accuracy. The tradeoff between the no-state-loss technique and the fill-flush technique is between speed of evaluation and accuracy. No-state-loss appears to be 2-3 times slower than fill-flush. However, no-state-loss is approximately twice as accurate for smaller caches than the fill-flush approach and has perfect accuracy for large caches. Another compelling reason to use no-state-loss sampling for single-pass algorithms is that extensions (such as the multiprogramming extension) are not compatible with fill-flush or other traditional sampling approaches since the stack is not maintained between samples.

A new model for involuntary context switching was developed based on distribution of the interval between a reference to an address and its re-reference. Methods were developed to integrate this model and a corresponding metric for voluntary context switches into the single-pass method. The new single-pass method was verified empirically and the behavior of the model was probed using the benchmarks.

The usefulness of the sampling-based, single-pass method extended to include multiprogramming effects was demonstrated by deriving prototypes for data cache systems from a broad design space of 324 possible cache designs per benchmark.

4. PROCESSOR PROTOTYPING

A processor design must execute instructions in a highly efficient manner while satisfying technology constraints such as pin count, clock frequency and overall transistor count. A successful prototype procedure for the processor must include the hardware resources that the instruction stream demands while not wasting VLSI real estate and endangering the cycle time by over-designing. This chapter presents two methods for processor prototyping. One method is based on the resource requirements of benchmarks. This method supplies an unlimited amount of hardware resources to the instruction stream while fixing technology parameters such as issue rate and function unit latency. The measured amount of resource usage from this unlimited-supply simulation is used to form a prototype processor design. Another method uses a cost-based exploration of a large space of possible processor configurations. This second method is well-suited for situations in which the function unit latencies are not fixed by technology. Empirical results are presented to validate these methods.

The processor prototyping techniques must be supported by efficient and high-speed simulation methods. A portion of this chapter is devoted to the discussion of simulation methods for superscalar processors. In order to extend the speed and trace-size range of the processor simulation algorithms, statistical sampling is adapted from a traditional cache simulation for use in processor simulation.

4.1 Processor Design Considerations

Figure 4.1 shows two instruction sequences, *(a)* and *(b)*. In the sequence *(a)*, in-

(a)	*(b)*
$I_1 : r1 \leftarrow \text{IAlu}(r2, r3)$	$I_4 : r1 \leftarrow \text{IAlu}(r2, r3)$
$I_2 : r4 \leftarrow \text{IAlu}(r5, r6)$	$I_5 : r5 \leftarrow \text{IAlu}(r1, r4)$
$I_3 : r7 \leftarrow \text{IMul}(r8, r9)$	$I_6 : r7 \leftarrow \text{IMul}(r5, r6)$

Figure 4.1: Two example instruction sequences.

structions can be executed in parallel. In sequence *(b)*, instructions I_5 must wait for instruction I_4 to complete since I_5 uses the value that I_4 defines (i.e., $r1$). Similarly, I_6 must wait until I_5 executes before it can use the value stored in $r5$. Sequence *(a)* may run three times faster than *(b)* with a properly constructed processor that allows independent instructions to execute in parallel. Sequence *(a)* is therefore said to possess *instruction-level parallelism* [39].

The instruction stream input to the processor is a list of atomic instructions, I_j, of the form:

$$I_j : d_j \leftarrow u_j(s_{j0}, s_{j1}, \ldots), \tag{4.1}$$

where d_j is the destination for I_j, u_j is the function unit used by I_j, and s_{jk} are the source operands used by I_j. The operator $U(I_j)$ is defined such that

$$U(I_j) \equiv u_j, \tag{4.2}$$

which is equivalent to the *function type used by* I_j. The set $\mathcal{S}(I_j)$ is defined such that

$$\mathcal{S}(I_j) \equiv s_{j0}, s_{j1}, \ldots, \tag{4.3}$$

Where $\mathcal{S}(I_j)$ is equivalent to the *set of source operands to* I_j. The number of source operands of I_j is defined to be $|\mathcal{S}(I_j)|$. Both d_j and members of $\mathcal{S}$ may be registers of the form rX, or memory addresses, represented using lower-case Greek letters, to be consistent with the conventions of Chapter 3. There are no addressing modes other than register, memory, and literal. All other address calculations are explicitly performed by instructions. All side-effect register address calculations (e.g., *autoincrement*) are excluded from the processor architecture and the equivalent semantics are performed by explicit instructions. The class of processors described in this chapter has an unlimited supply of both integer and floating-point registers. However, memory operations are still performed as directed by the program, since all program variables reside in memory. A special register, `cc`, is the condition code register. Its value determines the behavior of conditional control transfer instructions. This instruction format is derived from the intermediate code of the GNU C retargetable compiler, as discussed in Chapter 2.

The dependencies between I_5 and I_4 and between I_6 and I_5 of Figure 4.1 are called *pure* or *flow dependencies*. A flow dependence occurs when the definition of a register's

contents is followed by a use of that register. Such dependencies are caused by the structure of the computation that the instructions perform. It is not possible for the processor to remove flow dependencies from an instruction stream and still produce the correct result. Using the traditional notation of dependence analysis, $I_j \delta I_k$, if the destination of I_j is one of the sources of I_k, and $j < k$ (i.e., $d_j \in \mathcal{S}(I_k)$) [40] .

Consider sequences *(c)*, and *(d)* shown in Figure 4.2. There are no flow dependencies

(c)	*(d)*
$I_7 : r1 \leftarrow \text{IAlu}(r2, r3)$	$I_{10} : r1 \leftarrow \text{IAlu}(r2, r3)$
$I_8 : r2 \leftarrow \text{IAlu}(r5, r6)$	$I_{11} : r5 \leftarrow \text{IAlu}(r1, r4)$
$I_9 : r5 \leftarrow \text{IMul}(r8, r9)$	$I_{12} : r1 \leftarrow \text{IMul}(r5, r6)$

Figure 4.2: An example of anti- and output dependencies.

in sequence *(c)*; however, I_8 cannot execute until after I_7 completes since I_8 redefines $r2$. Similarly, I_9 must wait until I_8 completes. This situation is called an *anti-dependence* [40]. Anti-dependencies are due to the reuse of storage locations, in this case, $r2$ and $r5$. An anti-dependence occurs when the use of a register is followed by a redefinition of the register's contents. A similar problem exists in sequence *(d)*, where both I_{10} and I_{11} must execute before I_{12}. In this case, both I_{10} and I_{12} define the same location, $r1$. This situation is called an *output dependence*, which occurs when a definition of a register is followed by a redefinition of the register. Both anti- and output dependencies can be removed by the processor. These dependencies are not an artifact of the computation being performed, rather they are due to register reuse. Anti- and output-dependencies are assumed to be removed via *renaming hardware*. Renaming hardware detects register reuse

Table 4.1: Function types.

Function unit	Description
IAlu	Integer ALU
IMul	Integer multiply
IDiv	Integer divide, remainder
AddrC	Address calculation
Move	Register to register move
Shift	Shift
Load	Load
Store	Store
FPAdd	FP add
FPMul	FP multiply
FPDiv	FP divide, remainder
FPCvt	FP convert
Branch	Branch
Test	Compare/test

and remaps the reused register names to unused registers. Renaming is not performed for memory locations.

An assumption must be made about the decomposition of function types supported by the instruction set. These types are implemented by one or more *function units* that perform the functions in hardware. The types used in this thesis are presented in Table 4.1. Function units of the *IAlu* type implement integer addition, subtraction, and binary logical operations such as AND, OR, and XOR. The instructions that perform address calculation for complicated addressing modes are executed by *AddrC* function units. In some processors, an *IAlu* function unit might be used for *AddrC* operations. This is also the case with *Move* function units that perform register-to-register move operations. Some processors implement register-to-register moves via an *IAlu* that performs

a null calculation (e.g., add zero to rX). *Shift* function units perform binary bitwise shift operations. These are often implemented using a barrel shifter. The *Load* and *Store* function units access the first-level data cache. Floating-point operations are performed by the function units *FPAdd* (floating-point addition), *FPMul* (floating-point multiply), *FPDiv* (floating-point divide), and *FPCvt* (conversion to- and from floating-point format). The *Branch* function unit performs a control transfer, and the *Test* function unit sets the condition code register, `cc`, based on the value of its source operands.

In instruction sequence *(a)* of Figure 4.1, both I_1 and I_2 use an *IAlu* function unit. If the processor has only one *IAlu* function unit, competition for this resource can limit exploitable instruction-level parallelism. Such a situation is a type of *resource conflict*. Resource conflicts result in an artificial dependence between instructions. Formally, a resource conflict may exist between instructions I_j and I_k if $u_j = u_k$. Whether or not resource conflicts occur depends on whether I_j and I_k begin execution together and whether the function unit u_j $(= u_k)$ is in short supply at the time (this is explained in more detail below). This thesis presents methods to select a prototype in order to minimize resource conflicts.

4.1.1 Classes of processors

This thesis focuses on a class of high-performance processors that exploit instruction-level parallelism. This class of processors is depicted in Figure 4.3. Instructions flow from left to right in the figure. The action of bringing instructions into the processor

is called *instruction issue*. Instruction issue is composed of two phases: *instruction fetch* and *instruction decode*. The instruction fetch phase fetches instructions from the memory system. It must decide which instructions to fetch. This is a non-trivial task due to branch instructions. The instruction fetch unit often employs some form of branch prediction, which is discussed below. The instruction decode phase determines what operation the instruction performs. It is assumed in this chapter that all instructions are encoded in the same fixed-length format. As instructions leave the instruction issue stage they are said to be *issued*. Some processors allow more than one instruction to be issued per clock cycle. The number of instructions issued per clock cycle is termed the *issue rate*.

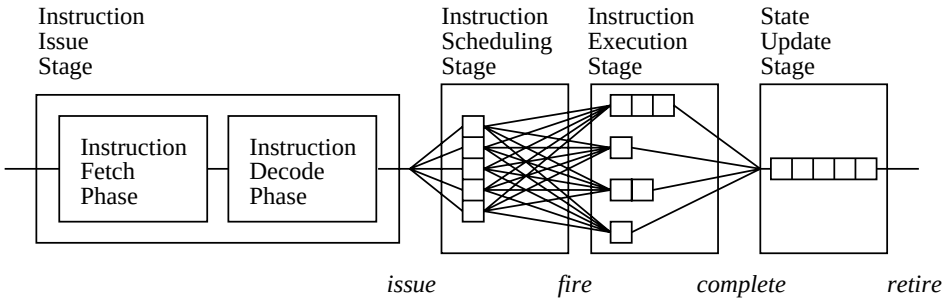

Figure 4.3: Class of processors considered in this thesis.

The stage that follows the instruction issue stage is the *instruction scheduling* stage. This stage determines whether instructions are dependent on or have resource conflicts with other instructions. Instructions that are independent and have no resource conflicts are passed onto the next stage for execution. It is assumed that all instructions spend at least one clock cycle in the instruction scheduling stage. The action of passing an instruction to execution is termed *instruction firing*.

The *instruction execution stage* takes instructions and performs the required computation by executing the instructions on one of many function units. The operator $N_U(u)$ is defined to be the *number of function units of type u* designed as parts of the execution stage of the processor. The function units can accept one instruction per cycle. If the operation that a function unit implements requires more than one cycle to execute, the function unit is pipelined into multiple stages and the pipeline is advanced at the rate of the system clock. The depth of the pipeline of a function unit determines the latency between the firing of the instruction and the completion of the instruction. This latency is referred to as the *function unit latency*. Typical function unit latencies are shown in Table 4.2. The latencies in this table are referred to as the *canonical latency set* for the remainder of this chapter.

Table 4.2: Canonical latency set.

Function unit	Typical latency
IAlu	1
IMul	3
IDiv	10
AddrC	1
Move	1
Shift	1
Load	2
Store	1
FPAdd	3
FPMul	3
FPDiv	10
FPCvt	3
Branch	1
Test	1

The operator $L(u)$ is defined to be the *latency of a function unit of type* u. For many complex functions, function unit latency is difficult to reduce past a practical limit. However, it may be cost-effective to increase the latency of a function unit that is little used. The topic of prototyping function unit latency is discussed in detail in Section 4.3.2. In accordance with previous stages, the action of instructions leaving the execution stage is termed *instruction completion.*

Figure 4.3 shows the instruction scheduling stage as a window that instructions enter from the instruction decode phase. The figure also shows a complicated crossbar interconnect between the instruction scheduling stage and the instruction execution stage. To reduce the cost of this interconnect, the window is often partitioned and the partitions are distributed to the function units. These distributed windows are typically called *reservation stations* [41]. This thesis does not consider in detail the decomposition of the scheduling window into reservation stations. The measurement of the required size of the scheduling window is discussed in Section 4.2.

After instructions have completed, they are passed to the final stage of the processor, the *state update stage.* This stage has as its responsibility maintaining a coherent sequential state in the register contents and for the memory system. This stage must also determine the correct instruction to begin executing after an interrupt occurs. The action of an instruction leaving this stage is termed *instruction retirement.*

The policy for instruction firing and instruction completion determines the class of a processor. A notation for these classes of processors can be constructed for these

Table 4.3: Processor classifications.

Notation	*F*iring policy	*C*ompletion policy
FICI	*I*n-order	*I*n-order
FICO	*I*n-order	*O*ut-of-order
FOCI	*O*ut-of-order	*I*n-order
FOCO	*O*ut-of-order	*O*ut-of-order

possible processors in the style of Flynn notation for parallel systems [42]. This classification is presented in Table 4.3. Fire in-order, complete in-order (FICI) processors are traditional pipelined processors. The FICO-class processors exploit some degree of instruction-level parallelism. The first machine to have a FICO-class processor was the Control Data 6600 [43]. (The CDC 6600 differed from the class of processors presented here since the function units were not pipelined.) The most-general class of processors is the FOCO; this class is the focus of this chapter. The first FOCO-class processor was implemented in the floating-point unit of the IBM 360/91, which had an issue rate of one instruction per cycle [41]. Using an FOCO-class execution model for all computation instead of restricting it to floating-point computation has been termed *superscalar* [44]. However, the term *superscalar* has also been used to describe FICO designs, since the CRAY-1 (a *supercomputer*) used a FICO policy [45]. Only FOCO-class designs are considered as superscalar processors in this thesis, and the term "FOCO-class processors," is used wherever possible in this chapter to reinforce this distinction.

The original implementation of FOCO-class processors allowed instructions to retire in the same order as they completed [41]. This presents a problem when an interrupt or

hardware exception occurs. Because instructions are allowed to fire and complete out-of-order, it might be the case that it is impossible to restart execution after an interrupt. This occurs when the registers and memory are left in a partial (i.e., non-sequentially consistent) state. Although this problem exists for FICO-class processors, its solution is simpler than what is required for FOCO-class processors [46],[47]. Techniques have since been developed to provide precise interrupts and coherent state for FOCO designs, including checkpoint-repair and history/reorder buffer approaches [47],[48]. This thesis assumes that instructions *always* retire in program order and that the processor uses either checkpoint-repair or a history/reorder buffer.

The final class of processors is the FOCI. The definition of this class implies that instructions are fired out-of-order yet forced to complete in-order. Such a regimen is impractical to build and offers no performance advantage over FICO- and FOCO-class processors. No FOCI-class architectures have been built to date.

4.1.2 Branch handling

The handling of conditional branch instructions in the processor is a topic that has received considerable attention. There are two separate issues in branch handing: (1) how to supply a sufficient number of the instructions from the branch target address to hide the latency of the non-sequential access in the instruction issue stage, and (2) how to predict the behavior of conditional branch instructions. This thesis assumes that a

mechanism exists to hide the non-sequential access, such as a branch-target buffer or forward slots, and does not discuss this mechanism in detail [49].

In the trace of the instruction stream, the behavior of all branch instructions is known. Using this trace without modification and assuming that branches are perfectly predictable are unrealistic. Consider the behavior of data-driven conditional branches that depend on data input from a file. Neither can hardware be constructed nor can any software be written to predict the behavior of these branches in all situations. Therefore, the issue of how to predict data-driven conditional branch behavior must be addressed in order to make simulation of the processor realistic.

Conditional branch behavior prediction schemes can be divided into hardware and software approaches. The majority of hardware schemes maintain information about branch behavior in a *history buffer*, which is a special-purpose cache [50],[51]. Software schemes use profile information to predict branch behavior [49].

This thesis simulates a branch prediction scheme that has high performance. A history buffer is used that is assumed to have unlimited entries. The scheme performs branch prediction as follows:

1. The first time that conditional branches are encountered, they are predicted to be not-taken, as is consistent with the suggestions in [51].

2. Two counters are kept in the buffer for each branch: n_T and n_B. The n_T counter is used to count the number of times a branch is *taken*. The n_B counter counts

the number of times a branch has been encountered up to the current point of execution.

3. If a conditional branch has an entry in the buffer and $n_T > n_B/2$, the branch is predicted *taken*; otherwise, it is predicted *not-taken*.

This is one of many possible hardware branch prediction schemes. The scheme does have acceptable performance. Table 4.4 displays the percentage of correct branch predictions for the benchmarks using the unlimited-entry history buffer (also shown is the geometric mean of these percentages). The scheme achieves its highest performance for the numeric members of the SPEC89 benchmark set (e.g., doduc, matrix300, nasa7 and tomcatv) and its lowest performance for a very data-driven benchmark (e.g., xlisp).

Table 4.4: Percentage of correctly predicted branches for the benchmarks.

Benchmark	Percentage
doduc	90.95%
eqntott	96.39%
espresso	85.08%
gcc	84.99%
matrix300	99.27%
nasa7	97.02%
tomcatv	95.23%
xlisp	71.21%
GMean	89.56%

4.1.3 Performance metrics

In the discussions that follow, several metrics are used to measure aspects of a processor's performance. One metric of performance is the *total execution time* for a benchmark, defined to be T_{TOT}. It is difficult to use T_{TOT} to summarize performance across several benchmarks since benchmark execution time varies widely. A better metric for cross-benchmark performance comparisons is *instructions per cycle* (IPC), the expected number of instructions in the execution stage during any cycle of execution. If there are N_I instructions executed in T_{TOT} time units, IPC = N_I/T_{TOT}. The IPC metric has been termed *instruction-level parallelism* elsewhere [44].

Metrics that measure the usage of function units and reservation stations are useful for prototyping since these metrics can be used to determine resource needs. The maximum amount of a resource in use at any clock cycle is one possible metric to use to determine resource needs (e.g., maximum number of *IAlu* function units used). This maximum is an upper bound, however, and the full distribution of the amount of the resource in use at every clock cycle provides more detailed information (See Section 4.3.1). The resource usage density function is defined as follows: let u be a resource from Table 4.1 (a function unit) or a window entry. In the case of window entries, $u \equiv W$. The simulation measures the function $T_u(n)$, the *total amount of time that exactly n resources of type u are active.* The resource usage density function, $f_u(x)$, is defined such that

$$f_u(x) \equiv \frac{T_u(x)}{T_{TOT}} \tag{4.4}$$

Equation (4.4) is an operational definition for $f_u(x)$. This definition also supplies an interpretation of $f_u(x)$: *the probability that x resources of type u are in use in any clock cycle during the simulation.* In some instances it is more appropriate to use cumulative distribution functions instead of density functions. The cumulative distribution function for $f_u(x)$, is defined such that

$$F_u(x) \equiv \sum_{x=0}^{x} f_u(x). \tag{4.5}$$

For each benchmark and each of the function unit types, the resource usage density functions for the function units ($f_u(x)$) and reservation stations ($f_u^{RS}(x)$) are measured. Table 4.5 summarizes the entire set of processor performance metrics.

Table 4.5: Processor performance metrics.

Metric	Description
T_{TOT}	Total execution time (in cycles)
N_I	Total number of instructions in the trace
IPC	Instructions executed per cycle, IPC = N_I/T_{TOT}
$f_u(x)$	Resource usage density function for resource u, where u is a function unit or W (the scheduling window)
$F_u(x)$	Resource usage cumulative distribution function for resource u

4.2 Processor Simulation Algorithms

This section discusses two methods to simulate FOCO-class processor systems. Each method has its advantages and disadvantages. The first method simulates the processor using three queues, one each for the instruction issue, instruction scheduling and instruction execution stages. It is a very detailed simulation, but lacks sufficient speed to be

useful in all cases. Statistical sampling is applied to this algorithm to improve its speed. The second algorithm also simulates realistic configurations, but it uses large amounts of memory during simulation. It is better suited for sampling applications. The purpose of this section is to introduce the algorithms, discuss the relevant tradeoffs of each, and to develop statistical sampling techniques for processor simulation.

4.2.1 Queue-based simulation algorithm

A queue-based FOCO-class processor simulation algorithm is shown in Figure 4.4. The algorithm maintains three queues, *issue, window*, and *exec*, which represent the instruction issue stage, instruction scheduling (window) stage, and instruction execution stage, respectively, of Figure 4.3. Operations on these queues are defined in Table 4.6.

Table 4.6: Operations on queues.

Operation	Description
add_to_queue(Q, x)	Add element x to the end of queue Q
delete_from_queue(Q, x)	Delete element x from queue Q
for $x \in Q$ **do**	Iterate over contents of queue Q
head(Q)	Return head of queue Q, leaving Q unchanged

For each instruction, the simulation maintains $N_{DEP}(I_j)$, *the count of the number of instructions* I_k *such that* $I_k \delta I_j$. In other words, $N_{DEP}(I_j)$ is the number of instructions I_j depends on. If $N_{DEP}(I_j) = 0$, the instruction is independent and is free to move from the *window* queue into the *exec* queue (Steps 6.4 and its substeps of Figure 4.4). $N_{DEP}(I_j)$ is calculated when an instruction enters the *window* queue by checking the source set, $\mathcal{S}(I_j)$, for any dependencies with instructions already in the *window* queue or the *exec* queue.

```
queue_based(I_j):
1       N_DEP(I_j) ← 0
2       for r ∈ S(I_j) do
2.1         I_k ← scoreboard[r]
2.2         if I_k ≠ ∅ then
2.2.1           L_DENT(I_k) ← L_DENT(I_k) ∪ {I_j}
2.2.2           N_DEP(I_j) ← N_DEP(I_j) + 1
3       add_to_queue(issue, I_j)
4       scoreboard[d_j] ← I_j
5       num_in_issue ← num_in_issue + 1
6       if num_in_issue = issue_rate then
6.1         for m ← 1 to issue_rate do
6.1.1           I_k ← head(issue)
6.1.2           delete_from_queue(issue, I_k)
6.1.3           add_to_queue(window, I_k)
6.2         T_TOT ← T_TOT + 1
6.3         for u ∈ function unit types do
6.3.1           num_fired[u] ← 0
6.3.2           num_exe[u] ← 0
6.3.3           for ℓ ← 0 to L(u) do stage_busy[u, ℓ] ← 0
6.4         for I_j ∈ window do
6.4.1           if N_DENT(I_j) = 0 and num_fired[U(I_j)] < N_U (U(I_j)) then
6.4.1.1             delete_from_queue(window, I_j)
6.4.1.2             add_to_queue(exec, I_j)
6.4.1.3             T_EXE(I_j) ← L(U(I_j))
6.4.1.4             num_fired[U(I_j)] ← num_fired[U(I_j)] + 1
6.5         for I_j ∈ exec do
6.5.1           stage_busy[U(I_j), T_EXE(I_j)] ← stage_busy[U(I_j), T_EXE(I_j)] + 1
6.5.2           num_exe[U(I_j)] ← max {num_exe, stage_busy[U(I_j), T_EXE(I_j)]}
6.5.3           if T_EXE(I_j) = 0 then
6.5.3.1             delete_from_queue(exec, I_j)
6.5.3.2             for I_k ∈ L_DENT(I_j) do N_DEP(I_k) ← N_DEP(I_k) − 1
6.5.3.3             if scoreboard[d_j] = I_j then scoreboard[d_j] ← ∅
6.5.4           T_EXE(I_j) ← T_EXE(I_j) − 1
6.6         for all u do T_u (num_exe[u]) ← T_u (num_exe[u]) + 1
7       num_in_issue ← 0
8       N_I ← N_I + 1
```

Figure 4.4: Queue-based FOCO-class processor simulation algorithm.

This is done by maintaining an array, *scoreboard*, that is used to reserve a destination register by an incoming instruction (Step 4). This method of using *scoreboard* simulates the action of renaming hardware since prior reservations for a register are overwritten in Step 4. When a dependency, $I_j \delta I_k$, is found, $N_{DEP}(I_j)$ is incremented by one (Step 2.2.2). So that this dependency can eventually be cleared, instructions also have associated with them a *list of dependent instructions*, defined as L_{DENT}. In this example, $L_{DENT}(I_k)$ will have I_j added to it (Step 2.2.1). The counter, $T_{EXE}(I_j)$, represents *the total number of cycles that I_j must spend in the execution stage*. This counter is set to the correct unit latency when I_j is placed in the *exec* queue (Step 6.4.1.3).

For each simulated clock cycle, the *exec* queue is scanned after the *window* queue (Steps 6.4 and its substeps). To count the number of function units in use at any one time, an array, stage_busy$[u, \ell]$, is maintained. This array is indexed by function unit type, u, and the value of T_{EXE} for an instruction in the *exec* queue. The array is initialized to 0 in Step 6.3.3. It is used to count the number of function units in each stage of the pipelined function units of type u. The number of active function units of type u, *num_exe*$[u]$, is determined by finding the maximum number of stages occupied (Step 6.5.2). This method is illustrated in Figure 4.5. The figure shows two scenarios for three instructions that use the same function unit type. In scenario A, the values of T_{EXE} for the three instructions are not the same (i.e., the three instructions were fired in different cycles of execution). The values of *stage_busy* just after the execution of Step 6.5.1 are shown below each diagram. In scenario B, $T_{EXE}(I_1) = T_{EXE}(I_2)$ (i.e., I_1 and

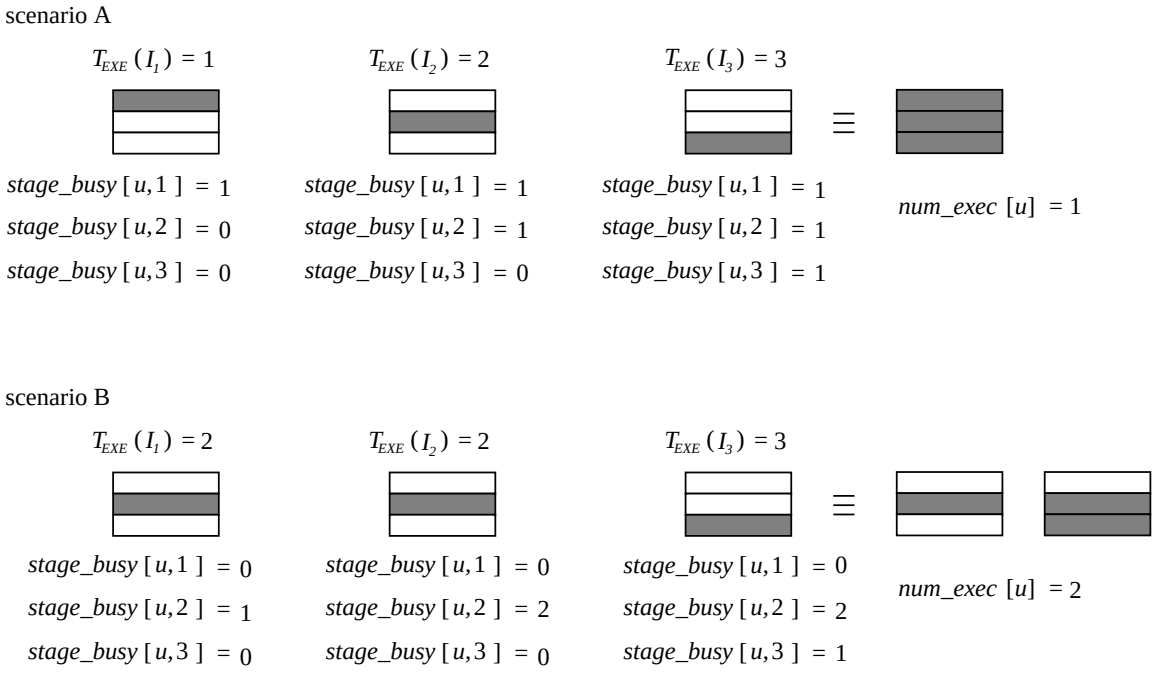

Figure 4.5: Two scenarios to explain the use of *stage_busy* in the queue-based simulation algorithm.

I_2 were fired in the same cycle), forcing the need for two function units. The equivalent function unit status is shown to the right of scenario B. The other permutation of this status is also valid. The $num\_exec[u]$ counters are used to update the $T_u(x)$ metrics, which in turn are used to calculate the $f_u(x)$ metrics using Equation (4.4).

Function unit latency is simulated in the algorithm by decrementing $T_{EXE}(I_k)$ for instruction I_j (Step 6.5.4). When $T_{EXE}(I_j) = 0$, I_j completes (Steps 6.5.3 and its sub-steps). A completed instruction is removed from the execution queue (Step 6.5.3.1). Upon completion of I_j, each entry of $L_{DENT}(I_j)$ has its N_{DEP} counter decremented by one (Steps 6.5.3.2 and 6.5.3.2.1). Instructions that have $N_{DEP} = 0$ due to this decrement

operation enter execution at the next simulated clock cycle. The reservation on for the destination register of I_j is removed if I_j still owns the reservation (Step 6.5.3.3).

4.2.2 Event-driven simulation algorithm

The queue-based simulation algorithm provides a large amount of information, but at the cost of requiring the simulation of each cycle of the benchmark's execution. This section presents an algorithm that does not need to simulate each clock cycle but instead derives the required information from the structure of the instruction trace.

For each instruction, it is possible to define four event times:

t_I: The *issue time*.

t_δ: The *dependence time*, the earliest an instruction can fire due to dependencies with other instructions ($t_\delta \geq t_I$).

t_{RC}: The *resource conflict time*, the earliest an instruction can fire due to resource conflicts with other instructions ($t_{RC} > t_\delta$).

FT(I_j): The actual *fire time of* I_j.

In the absence of other restrictions, FT(I_j) = t_{RC}. The definitions are such that $t_I \leq t_\delta \leq t_{RC}$.

Figure 4.6 shows an algorithm for calculation of the four event times for some I_j. As with the presentation of the queue-based simulation algorithm, the event-driven algorithm is presented for one instruction from the trace. In practice, the procedure *event_driven()* is applied sequentially to every instruction in the trace.

```
event_driven(I_j):
1       N_I ← N_I + 1
2       if (N_I mod issue_rate) = 0 then
2.1             t_I ← t_I + 1
3       t_δ ← t_I
4       for r ∈ S(I_j) do
4.1             I_k ← scoreboard[r]
4.2             if I_k ≠ ∅ then
4.2.1                   t_δ ← max {t_δ, FT(I_k) + L(U(I_k))}
5       scoreboard[d_j] ← I_j
6       t_RC ← t_δ + 1
7       success ← false
8       while ¬success do
8.1             if num_fired[U(I_j), t_RC] < N_U(U(I_j)) then
8.2.1                   num_fired[U(I_j), t_RC] ← num_fired[U(I_j), t_RC] + 1
8.2.2                   success ← true
8.4             else
8.4.1                   t_RC ← t_RC + 1
9       FT(I_j) ← t_RC
10      T_TOT ← max {T_TOT, FT(I_j) + L(U(I_j))}
```

Figure 4.6: The event-driven FOCO-class processor simulation algorithm.

The issue time, t_I, is calculated by maintaining an instruction counter and incrementing t_I once an issue period (Steps 1–2.1 of Figure 4.6). Initially, the dependence time is set to the issue time. The maximum of the (already calculated) fire time plus the function unit latency (i.e., the *completion time*) of each member of $\{I_k | I_k \delta I_j\}$ determine the earliest time that I_j can fire due to a dependence. This time is calculated in Step 4 and its substeps in the figure. The array *scoreboard* is used in the same manner as in the queue-based algorithm to detect destination register reservations. Register renaming is implemented in Step 5. Initially, the resource conflict time, t_{RC}, is set to one plus the dependence time. The addition of one to the dependence time reflects the assumption mentioned in Section 4.1.1 that every instruction must spend at least one cycle in the instruction scheduling stage.

The **while** statement of Steps 8 and its substeps in Figure 4.6 are used to calculate the resource conflicts for I_j. This is done by maintaining the array $num\_fired[u, t_{RC}]$, which records *the number of instructions fired to function unit u at time t*. If $num\_fired[u, t_{RC}] < N_U(u)$, then there are sufficient function units remaining at time t to execute an instruction with potential resource conflict time t_{RC}. The **while** statement iteratively tries successive potential resource conflict times until this condition is met. When it is, *success* is set to **true** and the loop exits. (It is possible to calculate $T_u(x)$ from the $num\_fired[u, t]$ array; however, this calculation is not shown in the figure.) When the last instruction is processed by the algorithm, the value of T_{TOT} calculated in Step 10 of the figure is the run time of the entire trace.

It was mentioned above that the event-driven algorithm has some difficulties. These difficulties center around the $num\_fired[u,t]$ data structure, which can grow to the size of the trace. In the queue-based algorithm, all data structures used to represent an executing instruction can be purged after the instruction completed. This is not the case with the event-driven algorithm. The following section discusses a method that can be used in conjunction with the event-driven algorithm to overcome these problems.

4.2.3 Statistical sampling of instruction traces

A benchmark that runs for 90 seconds on an architecture that supports 20 million instructions/second executes approximately 1.8 billion instructions. The time required to simulate using this benchmark is too great to be tractable. To avoid this problem, studies of instruction-level parallelism tradeoffs have frequently used the first four to 10 million instructions of a benchmark [47],[52]. Such results are of questionable value since the first few million instructions might only capture the initialization phase of the benchmark and not the portion of the benchmark that performs actual work. A method of improving the quality of the simulation results is to use statistical sampling. Chapter 3 discusses the use of statistical sampling to improve the speed of single-pass algorithms. Statistical sampling is also applicable to FOCO-class processor simulation. This section investigates the application of sampling to the two simulation algorithms discussed above.

The same notation is used here as was used in Chapter 3. Statistical sampling takes N_S samples of length L_S instructions from the trace. Each sample is a *contiguous* block

of L_S instructions. These samples are applied to the simulator in the order they are taken from the trace. For this section, $N_S = 40$. There are L_G instructions between each sample. L_G is adjusted so that the majority of the benchmark's dynamic execution is captured.

The measurement of the accuracy of the sampling approach is done by comparing IPC of the sampled runs to the IPC from the whole trace. For the miss ratio, ρ, in Chapter 3 it was guaranteed that $\rho < 1.0$ for all benchmarks. Unlike ρ, IPC is not bounded above by some value across all benchmarks. Relative error naturally normalizes the differences so that RE(IPC) < 1.0 for all benchmarks. This allows for cross-benchmark comparisons using one metric. Relative error is used here. It should be noted, however, that the errors presented in this section are not of the same magnitude as the errors presented for the cache simulation sampling scheme.

Three benchmarks are used for the study of sampling: doduc, espresso and gcc. The queue-based processor simulation algorithm is used since it does not have restrictions on the size of its internal data structures. The size of the window of the instruction scheduling stage and the number of function units are left unbounded in order to achieve a large amount of state in the simulation algorithm. An issue rate of eight instructions/cycle is used since this high issue rate allows a large number of instructions to execute in parallel. This produces a complicated state that is difficult to recreate, placing a large strain on the sampling process. The processor used has an unlimited supply of function units and window entries. The latencies assumed are those of Figure 4.2.

The sample size is difficult to select since there is no published previous work on the topic of sampling for instruction traces. Simulations must be performed to investigate sample size. The shorter lifetime of instructions in a processor compared to the lifetine of references in a cache system suggests that shorter sample sizes can be used for processor simulation than for memory system simulation. Figure 4.7 presents the relative error of the IPC versus sample size for gcc, espresso, and doduc with $2,000 \leq L_S \leq 20,000$. The vertical scale of the figure has been halved in order to show sufficient detail. The reader is reminded that this is relative error, and, as such, an error of 10% is quite acceptable. The sampling is performed such that sample runs with samples of size L_S contain sample runs of size less than L_S as prefixes to each sample. For example, the fifth sample taken for $L_S = 4,000$ begins with the fifth sample taken for $L_S = 3,000$ and includes 1,000 additional instructions. This is done to guarantee that the differences between accuracies are due to extending sample lengths and not a byproduct of sampling different regions of the trace. The drawback to this approach is that the accuracy of the sampling will not be perfect if the largest sample size considered does not have perfect accuracy.

The curves presented in of Figure 4.7 appear to have asymptotic behavior. The knee all three curves occurs for L_S in the range of 10,000–20,000. The sampled IPC for gcc has the best performance, displaying an optimal sample size of approximately $L_S = 6,000$. Espresso appears to have constant error beyond approximately 10,000–15,000. Espresso is also interesting because several very small sample sizes have low error. Increased sample size must ultimately result in higher accuracy since the limiting case is to use the

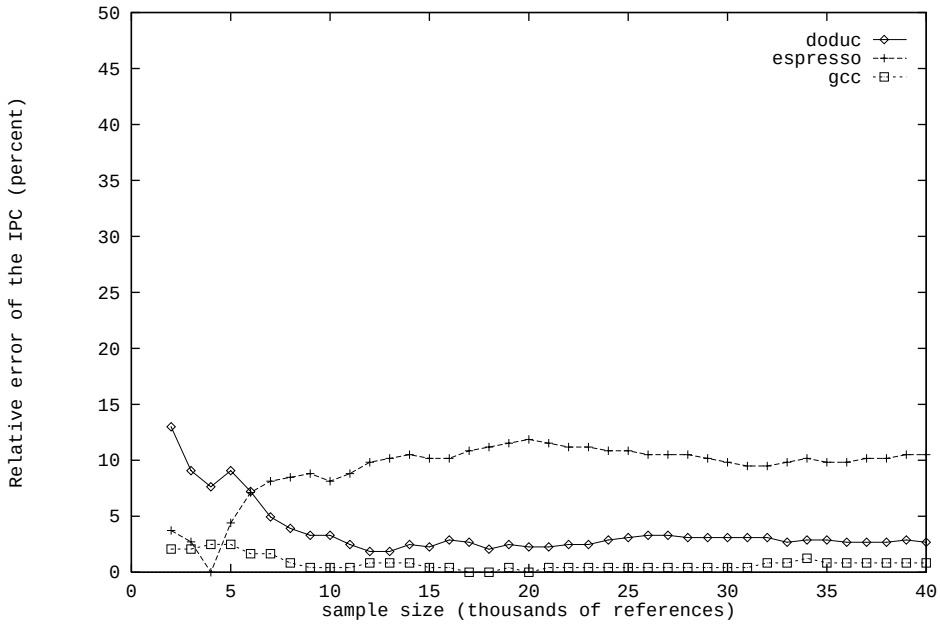

Figure 4.7: RE(IPC) versus sample size for gcc, doduc, and espresso, issue rate eight instructions per cycle.

entire trace for simulation. It is fortunate that a wide range of sample sizes has been considered for several benchmarks; otherwise, it would have been incorrectly concluded that $L_S = 4,000$ is an ideal sample size for all benchmarks.

Table 4.7 presents the relative errors for gcc, doduc, and espresso for issue rates of two, four and eight instructions per cycle, using sample sizes of $L_S = 10,000$ and $L_S = 20,000$. The table confirms that relative error increases with issue rate. Sample sizes of $L_S = 20,000$ perform as well as or better than the smaller sample sizes of $L_S = 10,000$, except for gcc. Figure 4.7 suggests that this is due to random fluctuations of sample error for gcc. The espresso benchmark remains the most-difficult to capture with sampling, with the relative error of 11.9%. This error is considered relatively minor,

however, and the data suggest that sample sizes of $L_S = 10,000$ produce acceptable error, with $L_S = 20,000$ providing some additional accuracy in certain situations.

Table 4.7: RE(IPC) for gcc, doduc, and espresso, for $L_S = 10,000$ and $L_S = 20,000$, issue rates two, four and eight instructions per cycle.

	issue rate = 2		issue rate = 4		issue rate = 8	
Benchmark	10,000	20,000	10,000	20,000	10,000	20,000
doduc	0%	0%	0.96%	0.64%	3.30%	2.27%
espresso	4.32%	4.32%	8.19%	8.30%	8.14%	11.9%
gcc	2.01%	1.34%	2.88%	1.92%	0.41%	0%

An alternative to reducing the sampling error would be to increase the number of samples taken from the trace [53]. This approach is consistent with the suggestions of [5]. The rationale is that the instruction samples have many interdependencies and as such are likely to not be normally distributed. This extension of processor sampling to more than 40 samples is suggested as future work and not investigated here.

The resource usage density functions are also an important metric to capture using sampling. The sampling process captures these quite well. Figure 4.8 presents the tabulated results of the full-trace distribution and the sampled distribution for the case of espresso with $L_S = 20,000$, issue rate of eight instructions per cycle (the case with the largest IPC error). The rows with primed labels (e.g., $IAlu'$) are for the sampled density functions and the unprimed rows are for the full-trace functions. In some situations in the figure, there are some differences of 2–3%, but the shapes of the distributions have been preserved. This figure provides evidence that the sampling process is recording the resource usage of the whole trace accurately.

Figure 4.8: Full-trace vs. sampled resource usage density functions for espresso, $L_S = 20,000$, issue rate eight instructions per cycle.

	Number of units active														
Type	0	1	2	3	4	5	6	7	8	9	10	11	12	13	≥ 14
IAlu	57	24	10	3	2	0.7	0.3	*	*	*	*	*	*		
IAlu′	60	25	9	2	1	0.5	0.1	*	*	*	*	*	*	*	*
IMul	99	*	*												
IMul′	99	0.1	*												
IDiv	100														
IDiv′	99	*	*	*	*										
AddrC	68	14	9	5	1	0.2	0.1	*	*	*	*		*		
AddrC′	70	13	9	4	1	0.2	0.1	*	*	*	*	*	*	*	*
Move	86	11	1	0.3	0.1	0.1	*	*							
Move′	87	9	1	0.4	0.1	0.1	*	*	*	*					
Shift	76	20	1	1	0.3	0.2	*		*	*					*
Shift′	76	20	1	1	0.2	0.2	*	*	*	*	*	*	*		*
Load	45	19	18	9	4	1	0.3	0.1	*	*	*	*	0.1	0.2	0.3
Load′	48	20	16	7	4	1	0.3	0.2	0.2	*	0.1	*	0.1	0.1	0.2
Store	82	7	9	0.7	0.5	*	*	*	*						
Store′	87	5	6	0.5	0.4	*	*	*	*	*	*	*	*	*	*
FPAdd	100														
FPAdd′	100														
FPMul	100														
FPMul′	100														
FPDiv	100														
FPDiv′	99	*													
FPCvt	100														
FPCvt′	99	*													
Branch	70	24	3	0.8	0.3	0.3	*	*	*	*					
Branch′	72	23	2	1.0	0.2	0.2	*	*	*	*		*			
Test	91	7	0.6	*	0.1	*									
Test′	91	8	0.6	*	*	*	*	*	*						

The simulations reported in Figure 4.7 and Table 4.7 do not account for lost state between samples, instead the processor state is flushed between each sample. Section 3.2.2 discussed methods for state repair of cache simulation. State repair is less of a problem in processor simulation due to the shorter lifetime of instructions. Consequently, less sophisticated state repair techniques are suggested for processor simulation. Specifically, the state repair problem is addressed by using some fraction of the sample to prime the processor state and delay recording of performance metrics until after this priming is completed. The string of priming instructions has length h. This is the analogue of the "cache warm-up" approaches of Section 3.2.2.

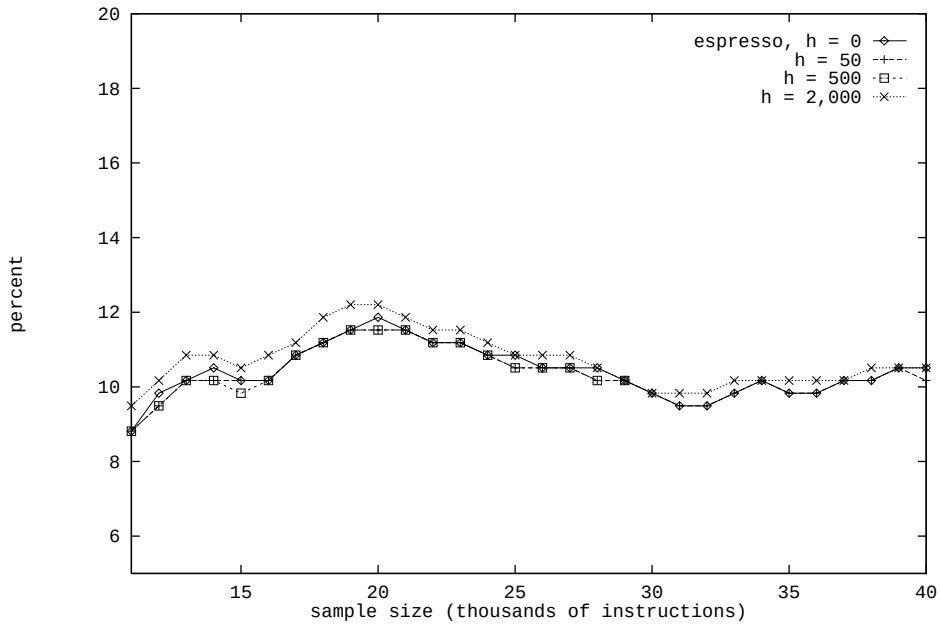

Figure 4.9: Effect of state repair on RE(IPC) for espresso, $h = 0$, 50, 500, and 2000.

Figure 4.9 demonstrates the value of using state repair for processor sampling. The figure shows the relative error of the IPC for the espresso benchmark using $h = 0$, 50,

500, and 2000 with sample sizes in the range $10,000 < L_S \leq 40,000$. The vertical scale in the figure has been exaggerated to show details between the schemes. The figure demonstrates that large values of heatup, such as $h = 2,000$, are detrimental to the accuracy of the sampling process. Small values of heatup, such as $h = 50$ and $h = 500$, have only marginal effect. The conclusion that can be drawn is that state repair of this type is not effective in improving the performance of the sampling technique. The data also suggest that state repair is not an issue for processor simulation.

These data can be viewed as a recommendation for how to sample instruction traces. The sample size should be approximately 10,000 to 20,000 samples. Flushing the state of the processor in-between samples is reasonable and the entire sample should be applied to the simulation without attempting state repair. Although the queue-based algorithm was used above, the event-driven algorithm should perform equally well with sampling, since both algorithms simulate the same processor model. The experimental results presented in the remainder of this chapter and those presented in the next chapter use this sampling regimen.

4.3 A Prototyping Approach to FOCO-Class Processor Design

Due to their complexity, FOCO-class processors are difficult and costly to design and implement. The prototyping of memory systems that was presented in Chapter 3 focused on capturing the performance of a broad design space of memory systems. Once that space of performance numbers was gathered, the prototyping process reduced to

deciding between prototypes based on this performance and an unspecified cost function. The same approach could be used for processors, but the design space for processors is too large to allow such an exploration. Consider a processor that uses the function units given in Table 4.1. Assume that the goal of prototyping is to determine the number and latency of each function unit type. If the number of function units cannot exceed N_{FU}, the number the possible latency values are in the range $[1..L_{\max}]$, and there are M function unit types, then the number of designs to explore is

$$M^{L_{max} \times N_{FU}}. \tag{4.6}$$

For $M = 16$, $L_{max} = 10$, and $N_{FU} = 2$, the number of possibilities is approximately 1.2×10^{24}. This space is too large to explore exhaustively. It must be shortened and intelligently explored. The first half of this section develops such an exploration technique for situations in which the latencies of the function units are fixed. A selection criterion based on the distribution of resources from unlimited-resource simulation are evaluated. Prototypes are derived using these criteria. Another method is developed that uses a cost-based exploration of the design space. This second method is well-suited for situations in which the function unit latencies are variable.

4.3.1 Resource requirements

The goal of prototyping is to specify the architectural resources that a benchmark or set of benchmarks needs. One method of measuring resource needs is to supply an unlimited amount of resources and observe resource use. This method will not work if

the benchmark's needs are insatiable. The expected amount of instruction-level parallelism has been found to be large, yet limited, for particular workloads [52]. Since the issue rate is largely a technology parameter, it is fixed for each prototype. A finite issue rate ultimately limits the amount of parallelism that can be uncovered [54]. Because the parallelism is limited, assuming unlimited resources should prove to be a valid approach to establishing prototype processors. This conjecture is substantiated below with experimental results.

When an unlimited number of function units is supplied, the resource usage density functions, $f_u(x)$, provide an indication of how much of the resource is required by the benchmark. Examples of these density functions are presented in Table 4.8 for the doduc benchmark and Table 4.9 for the gcc benchmark. These values were derived from a queue-based simulation that assumed an issue rate of eight instructions per cycle with function unit latencies taken from the canonical latency set.

The tables confirm the prediction that actual function unit usage is large but bounded, even at a high issue rate. The density functions have relatively long tails (e.g., consider the density function for *AddrC* for either benchmark). The criterion for selecting the maximum number of function units used would be very expensive in practice. The criterion that is used is to select the smallest n such that $F_u(n) \geq z$, where z is some chosen level. For example, setting $z = 0.99$ would select the smallest n such that 99% of the execution time was occupied by cycles in which at most n units were in use. This is also done for the window size using $F_W(u)$ to select the size of the scheduling window.

Table 4.8: Function unit resource usage density functions (percentage) for doduc, issue rate eight instructions per cycle, canonical latencies.

	Resource usage distribution for function units ($f^{FU}(x)$, in percent)														
Type	0	1	2	3	4	5	6	7	8	9	10	11	12	13	≥ 14
IAlu	79	13	5	1	0.4	0.2	0.2	*	*						*
IMul	97	2	*	*											
IDiv	99	0.3													
AddrC	50	20	15	9	3	0.7	*	*	*	*	*	*	*		*
Move	69	21	6	2	0.8	0.2	*	*	*	*					0.2
Shift	82	13	3	0.6	*	*									
Load	29	25	22	14	5	2	0.8	0.3	0.2	*	*	*	*	*	*
Store	71	17	7	1	1	0.1	0.1	*	*	*	*	*			
FPAdd	46	34	15	2	0.3	*		*				*		*	0.6
FPMul	52	29	12	4	0.7	0.2	*	*				*			
FPDiv	58	36	4	0.4	0.2										
FPCvt	86	8	4	0.1											
Branch	76	17	5	0.8	*			*						*	0.2
Test	91	7	0.9	*	*			*							

(* less than 0.1%)

Table 4.9: Function unit resource usage density functions (percentage) for gcc, issue rate eight instructions per cycle, canonical latencies.

	Resource usage distribution for function units ($f^{FU}(x)$, in percent)														
Type	0	1	2	3	4	5	6	7	8	9	10	11	12	13	≥ 14
IAlu	66	17	6	7	2	0.1	*	*	*						
IMul	98	1													
IDiv	99	0.6	*												
AddrC	82	10	4	1	0.2	*	*	*	*	*	0.1	*		*	*
Move	90	7	1	0.4	0.2	*	*	*							
Shift	90	7	1	*	*	*			*						
Load	58	26	10	2	0.9	0.2	0.1	*	*	*					
Store	80	15	2	0.5	0.7	0.2	0.3	*	*	*	*		*		*
FPAdd	100														
FPMul	100														
FPDiv	100														
FPCvt	100														
Branch	66	28	4	0.6	0.1	*	*	*							
Test	88	9	1	*	*										

(* less than 0.1%)

Figure 4.10 presents the geometric mean of the IPC metric where the number of function units was selected using the above criteria for values of $z = 0.80, 0.95$ and 0.99. Also presented for comparison in the figure is the unlimited-resource IPC. The figure shows very little difference between the IPC for an infinite supply of function units and the IPC with $z = 0.99$. For the criterion of $z = 0.80$, the IPC values drop by approximately 20% for issue rate eight instructions per cycle. This decrease is less acute for lower issue rates. The change in performance is monotonic and proportional to the selection criterion, z. If this had not been the case, a selection criterion based on resource usage would be meaningless.

Additional results of the IPC for various selection criteria are presented in Table 4.10. The table also demonstrates that the selection criteria are less critical for smaller issue rates. In some cases, such as an issue rate of two for doduc and eqntott, the criterion makes little difference.

The scheduling window size selected for the benchmarks is presented in Table 4.11. The window size requirements are quite small for xlisp and matrix300. The requirements for doduc are large, with 147 entries required for $z = 0.99$, issuing eight instructions per cycle.

Tables 4.12 through 4.20 present the prototype function unit needs for issue rates of two, four and eight instructions per cycle and criteria of $z = 0.80, 0.95$ and 0.99. Several observations can be made from the tables:

- Hardware requirements increase with issue rate across all benchmarks.

Table 4.10: IPC for function unit number selection criteria of $z = 0.80, 0.95, 0.99$ and unlimited resources ($z = 1.0$).

	issue rate = 2				issue rate = 4				issue rate = 8			
Benchmark	0.80	0.95	0.99	1.0	0.80	0.95	0.99	1.0	0.80	0.95	0.99	1.0
doduc	1.65	1.79	1.80	1.80	2.68	3.04	3.10	3.14	3.64	4.38	4.64	4.90
eqntott	1.84	1.85	1.88	1.88	3.20	3.43	3.47	3.49	5.56	5.56	5.88	6.06
espresso	1.51	1.57	1.59	1.59	2.01	2.34	2.38	2.44	2.24	2.79	2.94	3.13
gcc	1.28	1.47	1.49	1.49	1.71	1.99	2.06	2.07	1.85	2.22	2.42	2.46
matrix300	1.88	1.99	1.99	1.99	3.71	3.85	3.92	3.96	7.11	7.27	7.49	7.81
nasa7	1.87	1.97	1.98	1.98	3.55	3.66	3.74	3.89	4.95	5.12	5.31	7.35
tomcatv	1.93	1.97	1.97	1.98	3.80	3.88	3.88	3.90	6.44	7.31	7.44	7.55
xlisp	1.02	1.07	1.08	1.09	1.17	1.24	1.28	1.29	1.23	1.35	1.38	1.41
GMean	1.59	1.68	1.69	1.70	2.54	2.75	2.81	2.84	3.52	3.92	4.10	4.40
Percent	94%	99%	99%	–	89%	97%	99%	–	80%	89%	93%	–

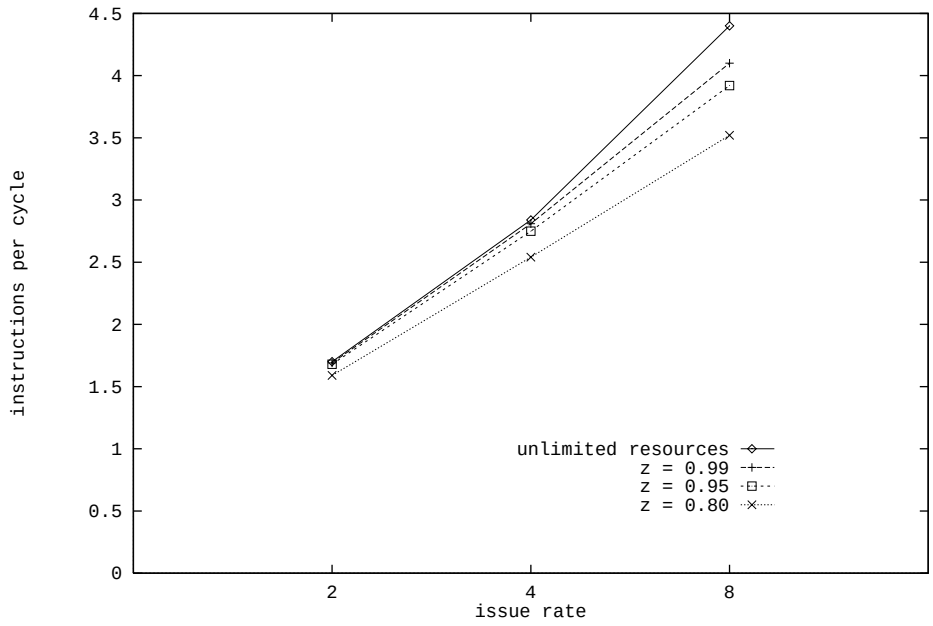

Figure 4.10: IPC for function unit number selection criteria of $z = 0.80, 0.95$ and 0.99 (geometric mean) for all issue rates, canonical latencies.

- The floating-point benchmarks do not have equivalent requirements. For example, doduc requires three *FPAdd*, four *FPMul*, two *FPDiv* and two *FPCvt* for issue rate eight and $z = 0.99$, whereas matrix300 requires only one of each (Table 4.20).

- All of the tables clearly indicate that *Load* (cache) ports are very important. Note that *Load* operations are not renamed like registers but some degree of parallelism can be obtained by issuing *Load* operations in parallel. Even using the criterion of $z = 0.80$ for an issue rate of two instructions per cycle suggests that two *Load* operations can be fired per cycle (Table 4.12).

- Other important function units are *IAlu* and *AddrC* operations.

- The four integer benchmarks do have similar characteristics: little or no floating-point hardware requirements, and similar *IAlu*, *IMul*, *IDiv*, *AddrC*, and *Shift* requirements. Significant differences do exist in the requirements for the memory interface function units. For example, eqntott requires nine *Load* operations for issue rate of eight instructions per cycle (Table 4.20).

- At most two to three Branch operations have to be executed in parallel across all benchmarks (with the exception of nasa7).

- In general, the function unit needs are moderate to support very high performance. At an issue rate of eight instructions/cycle, using the $z = 0.80$ designs (Table 4.18), the largest required number of function units of any one type is four (*IAlu* for eqntott). The geometric mean of the IPC for $z = 0.80$ is 3.52.

Table 4.11: Scheduling window size for selection criteria of $z =$ 0.80, 0.95, and 0.99.

	issue rate = 2			issue rate = 4			issue rate = 8		
Benchmark	0.80	0.95	0.99	0.80	0.95	0.99	0.80	0.95	0.99
doduc	22	55	84	47	90	112	74	112	147
eqntott	7	8	14	14	18	48	52	59	86
esp	10	15	24	21	63	100	38	87	112
gcc	5	11	21	20	58	93	48	68	110
matrix300	14	17	17	29	32	50	62	65	72
nasa7	15	21	26	53	57	65	73	71	84
tomcatv	17	31	42	44	68	82	79	97	116
xlisp	7	11	20	9	23	60	10	32	65

Table 4.12: Processor resource requirements, issue rate of two instructions/cycle, $z = 0.80$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	1	1	1	1	1	1	1
eqntott	1	1	1	1	1	1	1
espresso	1	1	1	1	1	1	1
gcc	1	1	1	1	1	1	1
matrix300	1	1	1	1	1	1	2
nasa7	1	1	1	1	1	1	1
tomcatv	1	1	1	1	1	1	2
xlisp	1	1	1	1	1	1	1
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	1	1	1	1	1	1	1
eqntott	1	1	1	1	1	1	1
espresso	1	1	1	1	1	1	1
gcc	1	1	1	1	1	1	1
matrix300	1	1	1	1	1	1	1
nasa7	1	1	1	1	1	1	1
tomcatv	1	1	1	1	1	1	1
xlisp	1	1	1	1	1	1	1

Table 4.13: Processor resource requirements, issue rate of two instructions/cycle, $z = 0.95$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	1	1	1	1	1	1	2
eqntott	1	1	1	1	1	1	1
espresso	2	1	1	1	1	1	2
gcc	2	1	1	1	1	1	1
matrix300	2	1	1	1	1	1	2
nasa7	1	1	1	1	1	1	2
tomcatv	2	1	1	1	1	1	3
xlisp	1	1	1	1	1	1	2
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	1	1	1	1	1	1	1
eqntott	1	1	1	1	1	2	2
espresso	1	1	1	1	1	1	1
gcc	1	1	1	1	1	1	1
matrix300	1	1	1	1	1	1	1
nasa7	1	1	1	1	1	1	1
tomcatv	1	2	2	1	1	1	1
xlisp	1	1	1	1	1	1	1

Table 4.14: Processor resource requirements, issue rate of two instructions/cycle, $z = 0.99$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	2	1	1	2	2	1	3
eqntott	1	1	1	1	1	1	2
espresso	2	1	1	2	1	1	2
gcc	2	1	1	2	1	1	2
matrix300	2	1	1	1	1	1	2
nasa7	2	1	1	2	1	1	2
tomcatv	3	1	1	2	1	1	4
xlisp	1	1	1	2	1	1	3
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	2	2	2	1	1	2	1
eqntott	1	1	1	1	1	2	2
espresso	2	1	1	1	1	2	1
gcc	2	1	1	1	1	2	1
matrix300	1	1	1	1	1	1	1
nasa7	2	2	3	1	1	1	1
tomcatv	2	4	2	1	1	1	1
xlisp	2	1	1	1	1	2	1

Table 4.15: Processor resource requirements, issue rate of four instructions/cycle, $z = 0.80$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	1	1	1	1	1	1	2
eqntott	2	1	1	1	1	1	2
espresso	1	1	1	1	1	1	2
gcc	1	1	1	1	1	1	1
matrix300	2	2	1	1	1	2	2
nasa7	1	1	1	2	1	1	2
tomcatv	2	1	1	1	1	1	2
xlisp	1	1	1	1	1	1	1
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	1	1	1	1	1	1	1
eqntott	1	1	1	1	1	1	1
espresso	1	1	1	1	1	1	1
gcc	1	1	1	1	1	1	1
matrix300	1	1	1	1	1	1	1
nasa7	1	1	1	1	1	1	1
tomcatv	1	2	2	1	1	1	1
xlisp	1	1	1	1	1	1	1

Table 4.16: Processor resource requirements, issue rate of four instructions/cycle, $z = 0.95$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	1	1	1	2	2	1	3
eqntott	3	1	1	2	1	1	3
espresso	2	1	1	2	1	1	2
gcc	3	1	1	1	1	1	2
matrix300	3	2	1	1	1	2	2
nasa7	1	1	1	1	1	1	2
tomcatv	3	1	1	2	1	1	4
xlisp	1	1	1	1	1	1	2
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	2	2	2	1	1	1	1
eqntott	1	1	1	1	1	1	1
espresso	2	1	1	1	1	1	1
gcc	1	1	1	1	1	1	1
matrix300	1	1	1	1	1	1	1
nasa7	1	1	1	1	1	1	1
tomcatv	2	4	4	1	1	1	1
xlisp	1	1	1	1	1	1	1

Table 4.17: Processor resource requirements, issue rate of four instructions/cycle, $z = 0.99$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	2	1	1	2	3	2	4
eqntott	3	1	1	2	2	1	3
espresso	3	1	1	3	2	2	4
gcc	4	1	1	2	2	2	3
matrix300	3	2	1	1	1	2	2
nasa7	2	1	1	2	1	1	2
tomcatv	5	1	1	3	1	2	8
xlisp	2	1	1	3	2	1	3
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	2	3	3	2	2	2	1
eqntott	1	1	1	1	1	2	1
espresso	2	1	1	1	1	2	1
gcc	2	1	1	1	1	2	1
matrix300	1	1	1	1	1	1	1
nasa7	2	2	3	1	1	1	1
tomcatv	2	4	4	1	1	1	1
xlisp	2	1	1	1	1	2	1

Table 4.18: Processor resource requirements, issue rate of eight instructions/cycle, $z = 0.80$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	1	1	1	2	1	1	3
eqntott	3	1	1	2	1	1	3
espresso	1	1	1	1	1	1	2
gcc	1	1	1	1	1	1	1
matrix300	3	2	1	1	1	2	3
nasa7	1	1	1	1	1	1	1
tomcatv	4	1	1	1	1	1	5
xlisp	1	1	1	1	1	1	1
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	1	1	1	1	1	1	1
eqntott	1	1	1	1	1	2	2
espresso	1	1	1	1	1	1	1
gcc	1	1	1	1	1	1	1
matrix300	1	1	1	1	1	1	1
nasa7	1	1	1	1	1	1	1
tomcatv	1	3	3	1	1	1	1
xlisp	1	1	1	1	1	1	1

Table 4.19: Processor resource requirements, issue rate of eight instructions/cycle, $z = 0.95$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	2	1	1	3	2	1	4
eqntott	3	1	1	2	1	1	3
espresso	3	1	1	3	1	1	3
gcc	3	1	1	2	1	1	2
matrix300	4	2	1	1	1	2	3
nasa7	1	1	1	1	1	1	2
tomcatv	6	1	1	2	1	2	7
xlisp	1	1	1	2	1	1	3
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	2	2	3	2	1	2	1
eqntott	1	1	1	1	1	2	2
espresso	2	1	1	1	1	1	1
gcc	1	1	1	1	1	1	1
matrix300	1	1	1	1	1	1	1
nasa7	1	1	1	1	1	1	1
tomcatv	2	5	4	1	1	1	1
xlisp	1	1	1	1	1	2	1

Table 4.20: Processor resource requirements, issue rate of eight instructions/cycle, $z = 0.99$.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	3	1	1	4	4	2	6
eqntott	5	1	1	2	3	2	9
espresso	5	1	1	4	2	3	5
gcc	4	1	1	3	2	2	4
matrix300	4	2	1	1	1	2	3
nasa7	2	1	1	2	1	1	2
tomcatv	6	1	1	4	2	2	8
xlisp	1	1	1	3	3	1	3
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	4	3	4	2	2	3	1
eqntott	1	1	1	1	1	3	3
espresso	3	1	1	1	1	3	2
gcc	4	1	1	1	1	2	2
matrix300	1	1	1	1	1	2	1
nasa7	2	2	3	1	1	1	1
tomcatv	4	5	4	1	1	2	1
xlisp	2	1	1	1	1	2	1

4.3.2 Near-optimal processor configurations

The above method for generating processor configurations from the usage of unlimited resources is useful to construct prototypes when altering the function unit latencies is not a design option. When function unit latency is also variable, the selection of prototypes can be carried out by performing an unlimited-resource simulation for each possible latency set. However, if the number of latency sets is too large for this approach to be feasible, one alternative is to formulate an optimization problem.

Consider a processor design space composed of one or more of the function units of Table 4.1, each having a latency ranging from 1 to $L_{\max}$. A processor, P, has n_j function units of type j and each of these function units has a latency of ℓ_j. A concise representation of processor m is

$$P = \langle (\ell_0, n_0), (\ell_1, n_1), \ldots, (\ell_{k-1}, n_{k-1}) \rangle, \tag{4.7}$$

for k different types of function units. Each design of P can be assigned a cost. The cost of a function unit u is defined to be $\mathcal{C}(u)$. The cost of the processor is then the cost of each of its function units times the number of each function unit, or,

$$\mathcal{C}_{TOT} = \sum_u k_u N(u) \mathcal{C}(u), \tag{4.8}$$

where k_u are constants of proportionality.

There are many factors that enter into the formulation of $\mathcal{C}(u)$, such as development time and complexity. It is more difficult to design an unpipelined function without also stretching the cycle time of the processor. Therefore, for the purposes of this section,

$\mathcal{C}(u)$ is defined to be one over the sum of the number of stages of function unit u. In terms of latency, $\mathcal{C}(u) = L(u)^{-1}$. Equation (4.8) can be rewritten as,

$$\mathcal{C}_{TOT} = \sum_{u} \frac{N(u)}{L(u)}, \tag{4.9}$$

where the k_u's have been set to unity. Equation (4.9) is the hardware cost model used below. It is not claimed that this model accurately reflects actual industrial design cost. Other cost models could be used. Equation (4.9) does, however, include the reluctance of designers to create numerous, low-latency function units.

The optimization problem is to minimize $\mathcal{C}_{TOT}(P)$ by changing the number and latency of the function units of P. The problem is sufficiently difficult so that a Monte Carlo technique is required. Of the Monte Carlo techniques, *Simulated Annealing* is chosen since it is well-suited to optimization problems [55],[56]. The version of the algorithm that follows is adapted from a simulated annealing algorithm for the *traveling salesman problem*, presented in [57].

The simulated annealing algorithm used here progresses from an initial processor design, P_0, to a final design, P_F, based on a *energy function*, $\mathcal{E}$ [55]. Since a computer is used to perform the optimization, the energy function can be quite complex. The following requirements are used to formulate the energy function for processor selection:

1. $\mathcal{C}_{TOT}(P_F)$ should be minimal, and

2. $\mathrm{IPC}(P_F) \geq 0.9 \times \mathrm{IPC}(P_0)$.

These two requirements are used to derive the following equation:

$$\mathcal{E}(P_i) = \begin{cases} \left(\frac{\mathrm{IPC}_{(P_0)}}{\mathrm{IPC}_{(P_i)}}\right) \mathcal{C}_{TOT}(P_i) & \text{if } \mathrm{IPC}(P_i) \geq 0.9 \times \mathrm{IPC}(P_0) \\ \infty & \text{otherwise.} \end{cases} \tag{4.10}$$

The ∞ guarantees that designs with IPC $< 0.9 \times \mathrm{IPC}(P_0)$ are never accepted by the algorithm.

Simulated annealing works by randomly altering P_i to create P_{i+1}, then evaluating the difference in the energy,

$$d\mathcal{E} = \mathcal{E}(P_{i+1}) - \mathcal{E}(P_i). \tag{4.11}$$

The algorithm decides to accept P_{i+1} if either:

- $d\mathcal{E} < 0$, or
- $rand\_num < e^{-d\mathcal{E}/t}$.

The number, *rand_num*, is a uniformly distributed random number in the range $[0, 1)$. The parameter, t, is the current *temperature*. Simulated annealing proceeds from an initial temperature, $t = t_0$, and reduces t by the fraction f_{STEP} at each step. At each step, the algorithm creates and evaluates $n_{\max}$ prototype processors, P_i. A total of n_{STEP} temperature steps are taken in total. For this thesis, these parameters are set as follows:

$$\begin{aligned} t_0 &= 0.05 \times \mathcal{E}(P_0), \\ f_{STEP} &= 0.1, \\ n_{\max} &= 96, \\ n_{STEP} &= 25. \end{aligned}$$

At each step of the algorithm, the design P_{i+1} is selected from P_i using a restricted random selection procedure. The following procedure for determining the latencies of P_i was found to produce acceptable results:

1. Select m latencies at random from P_{i+1}, where m is a random integer in the range $[1, 3]$.

2. For all $0 < k \leq m$, add to latency ℓ_k a random integer, x, in the range $[-4, 4]$. If $\ell_k + x < 0$ or $\ell_k + x > L_{MAX}$, reject the change to ℓ_k.

The same procedure is used to alter the number of the function units, n_k, for P_i.

The evolution of the energy function during the execution of the simulated annealing algorithm for gcc is presented in Figure 4.11. The initial design for the processor was taken from Table 4.20. The IPC for the initial design is 2.41. The hardware cost of this design is $\mathcal{C}_{TOT}(P_0) = 48.07$. The final design after simulated annealing is shown in Table 4.21. This processor has a performance of IPC = 2.21. The cost of the annealed result is $\mathcal{C}_{TOT}(P_F) = 7.46$, a 6.4 times reduction.

The algorithm is also used for the xlisp benchmark, and the results are presented in Table 4.22. For xlisp, IPC$(P_0) = 1.37$ with hardware cost $\mathcal{C}_{TOT}(P_0) = 16.03$. After annealing, IPC$(P_F) = 1.32$, with hardware cost $\mathcal{C}_{TOT}(P_F) = 6.65$, a 2.4 times reduction. It is interesting to note the algorithm selected includes only one function unit of each type for xlisp.

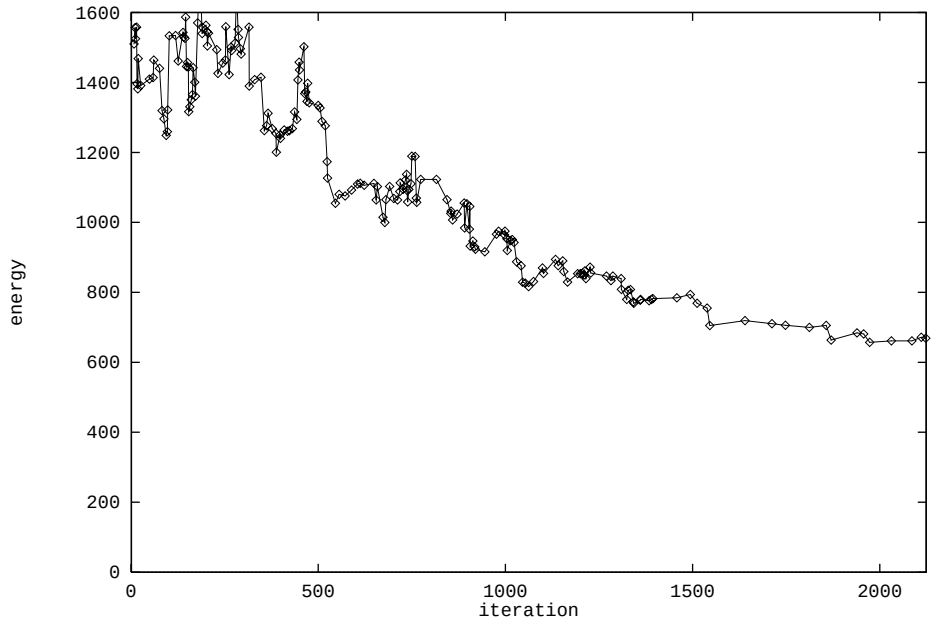

Figure 4.11: Evolution of energy for gcc benchmark during execution of simulated annealing algorithm.

Table 4.21: Final prototype for processor for gcc, issue rate eight instructions per cycle (IPC = 2.21).

Function unit	Number	Latency
IAlu	2	1
IMul	1	10
IDiv	1	6
AddrC	1	1
Move	1	2
Shift	1	4
Load	2	1
Store	3	6
FPAdd	1	4
FPMul	2	10
FPDiv	2	9
FPCvt	1	10
Branch	1	1
Test	1	1

Table 4.22: Final prototype for processor for xlisp, issue rate eight instructions per cycle (IPC = 1.32).

Function unit	Number	Latency
IAlu	1	2
IMul	1	8
IDiv	1	5
AddrC	1	1
Move	1	1
Shift	1	7
Load	1	1
Store	1	2
FPAdd	1	9
FPMul	1	8
FPDiv	1	8
FPCvt	1	5
Branch	1	1
Test	1	2

Consider the design decisions made by the algorithm for the gcc benchmark. In particular, the algorithm:

1. reduced the number of *IAlu* from four to two,

2. increased the latency of *IMul* to 10,

3. decreased the latency of *IDiv* to six,

4. reduced the needed number of address calculation units to one,

5. decreased the latency of *Move* function units to two

6. reduced the number of *Shift* function units from two to one and increased their latency to four,

7. reduced the number of *Load* function units from four to two and decreased their latency to one,

8. decreased the number of *Store* function units from four to three and increased their latency from one to six,

9. increased the latency and number of the floating-point function units, adding an additional *FPAdd* and *FPMul*, and

10. reduced the number of *Branch* and *Test* function units from two each to one each.

This list of modifications is extensive, suggesting that designing with fixed latencies is not an ideal situation. In practice, however, function unit design is a complicated task warranting design re-use between generations of processors. When the design space is much smaller, either the techniques of the previous section can be used, or a cost function expressing the limited design space can be incorporated into the annealing algorithm.

4.4 Summary of Processor Prototyping

The purpose of this chapter has been to present techniques to prototype FOCO-class processors. Two simulation techniques, the queue-based and the event-driven, were developed to explore possible processor designs. Statistical sampling was adapted from traditional cache simulation and applied to processor simulation. Prototype selection criteria based on the used resources from an unlimited-resource simulation were evaluated, found to be useful, and used to derive prototypes for the benchmarks. The general

optimization problem was evaluated using an adaptation of simulated annealing, and sample annealing schedules were presented.

5. PROCESSOR/MEMORY INTERACTION

Chapters 3 and 4 of this thesis have concentrated on the prototyping of the two primary components of a workstation: the memory system and the processor. It has been assumed that if both designs are high-performance, the interaction between the subsystems will not significantly change the designs of either. This assumption has to be verified. This chapter investigates the interaction between the processor and the memory system prototypes. A discussion of the relevant techniques that modern designs use to coordinate processor/memory interaction is presented. Empirical results are used to evaluate the techniques and demonstrate that sufficient design options exist to continue to justify separate designs for the processor and the memory system.

5.1 Processor and Memory System Interfaces

In Chapter 4, the *Load* and *Store* function units are described as the method for program interaction between the data portion of the memory hierarchy and the processor. It is assumed there that these operations have a fixed latency. The latencies presented,

however, are only valid if the data that are loaded or stored are present in the first-level data cache. When the data are not present, a resultant cache miss occurs and the correct data must be fetched from the second level in the memory hierarchy. This fetching operation is defined to take T_{MISS} time units. Methods to design the processor/memory interface to correctly handle cache misses is the primary topic discussed in this chapter.

There are several schemes to deal with data cache misses, this chapter discusses three schemes:

Blocking: In this scheme, the processor is *blocked* by halting the processor clock while the required data are fetched from the second level of the memory hierarchy. After the cache contents are updated, the clock is again applied to the processor. From the processor's point of view, the cache miss has the same latency as a cache hit. From the system users' point of view, however, each cache miss adds T_{MISS} cycles of execution.

Limited blocking: In this scheme, when a miss occurs, any subsequent access to the cache is frozen, as in the *blocking* scheme. The processor clock is not halted. Rather, the processor is allowed to continue to execute other instructions that do not access memory. If sufficient independent instructions are ready to execute, this scheme can effectively hide the performance impact of the cache miss. Limited blocking was suggested in [47].

Non-blocking: In this scheme, the cache is modified to update its contents on a cache miss while it continues to service requests for data that are present in the cache. Such

a design is termed a *non-blocking cache of degree n* if it can service n outstanding misses while continuing to service requests from the processor. When greater than n misses are outstanding, this scheme reverts to the *limited blocking* scheme. Non-blocking caches were introduced in [58].

Store instructions can be dealt with by writing the stored addresses to a *store buffer* that decouples the processor from the behavior of the stored operations [47]. When the *Store* operations are ultimately performed, if they generate a miss, then one of the three schemes above must be used. The processor is assumed to have a store buffer.

5.1.1 Blocking and limited-blocking cache designs

The modifications to the processor that must be performed to support the *blocking* scheme are minimal. The primary modification is to supply a separate clock to the processor from the clock that is used to advance the state of the cache. When a miss is detected in the cache, the clock to the processor is held low until the miss is repaired. If the processor contains states that involve dynamic memory elements, refresh signals must be provided to retain memory state.

A *limited-blocking* cache design does not require separate clocking schemes. These designs require flow control between the instruction scheduling stage of the processor and the cache controller. When a miss occurs, the cache controller signals the processor to halt cache accessing. This can be done by halting the advance of the *Load* and *Store* function unit pipelines. If the scheduling window is implemented via reservation stations,

once the *Load/Store* reservation stations are full, instruction issuing is halted and the processor reverts to the *blocking* scheme. After the miss is repaired, the cache signals the processor to resume. The *Load* and *Store* function unit pipelines then resume and begin making requests of the cache. The first request is to retry the missing access, which now finds the block present in the cache and results in a hit. The advantage of the *limited-blocking* scheme over the *blocking* scheme is that the processor can continue to execute non-memory-accessing instructions while the cache is repairing the missing block.

5.1.2 Non-blocking cache designs

The most complicated of the three schemes is the *non-blocking* scheme. A schematic of the design of a non-blocking cache is presented in Figure 5.1. The steps in the operation of

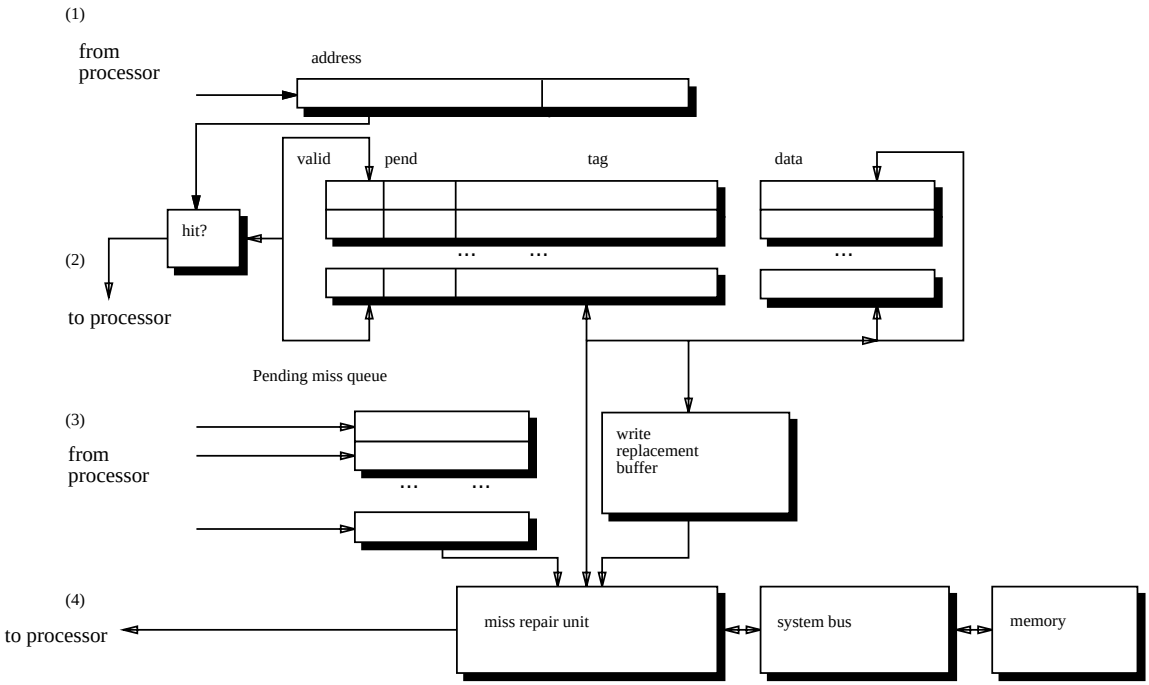

Figure 5.1: Schematic of the design of a non-blocking cache.

the cache are labeled (1) through (4) to the left edge of the figure. In Step (1), the address is transmitted to the cache from the instruction scheduling stage. The corresponding hit/miss signal is returned to the instruction scheduling stage in Step (2). Typically this can be done within one cycle. In the case of a hit, the data are fetched from the cache normally. In the case of a miss, the address is placed on the *pending miss queue* (Step 3). This structure is an FIFO queue containing address entries to fetch from the second level of the memory system. The *write replacement buffer* is used to hold contents of replaced dirty blocks in the cache. The *miss repair unit* performs the task of fetching the data from the second-level memory and writing it into the cache. Requests in the pending miss queue and the write replacement buffer are correctly ordered by the *miss repair unit*. This is done by use of a hardware timestamp generated by a recirculating counter. The miss repair unit inspects the requests at the top of the *pending miss queue* and the *write replacement buffer* and performs the earlier of the two requests. When the repair of the missing cache block is completed, the miss repair unit signals the instruction scheduling stage of the processor (Step (4)).

It is possible to design complicated hardware to combine entries in the *pending miss queue* with redundant entries in the *write replacement buffer*. Such hardware complicates the cache control logic of an already complicated cache design. The designs simulated in this chapter do not employ such combining hardware. These simplifications do not deter the performance of the *non-blocking* scheme, as is demonstrated in the experimental

results in Section 5.2. The state model for a pending miss in this design of a non-blocking cache is shown in Figure 5.2. This diagram illustrates that the state transitions of a pending miss are regular and uncomplicated.

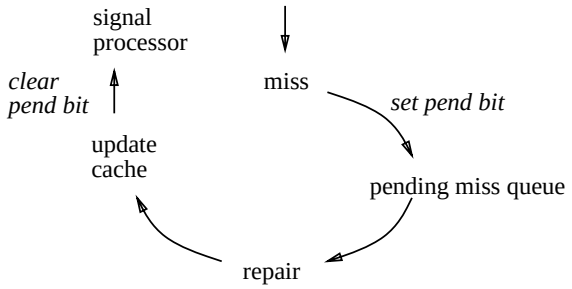

Figure 5.2: State model for a pending miss request.

In addition to the address being fetched from the second-level memory, the pending miss queue entries contain the name (e.g., Tomasulo-style *tag*) of the register being loaded for *Load* operations [58]. The load operation has an additional data dependence added to it. When the miss repair completes, this dependence is resolved and the load can fire. For *Store* operations, no register name is reserved in the pending miss queue. Instead, an index of an entry in the store buffer is retained. When the miss repair completes for the store, the store buffer contents at the corresponding index are used to update the status of the cache. (An alternative would be to implement a sectored cache with sector size equal to word size.)

The replacement policy of the cache can be altered if missed blocks are not allocated entries in the cache while they are being repaired. For example, least-recently used replacement can degenerate into random replacement. Instead of constructing complicated hardware to enforce the correct replacement scheme, the missing data are allocated a

cache block when the miss occurs. This is consistent with the philosophy stated previously of simplifying the cache control logic. To indicate that this block does not contain data, the block is marked as *pending* by use of an additional valid bit called the *pend* bit.

5.1.3 Performance metrics

The performance metrics for the study of processor/memory interaction are taken from the sets of performance metrics of the processor and of the memory system. The miss ratio, ρ, is used as the memory system metric. The miss ratio for the processor/memory combination is defined to be ρ^*. It is assumed that $\rho = \rho^*$ since the three schemes presented do not reorder memory accesses. The processor's performance is measured using the instructions per cycle of the processor/memory combination, represented as IPC*. The ratio of the the prototype processor IPC to IPC* is used to measure the degree of performance degradation due to the processor/memory interaction.

5.2 Empirical Evidence of Processor/Memory Interaction

This section addresses both the issue of the interaction between the processor/memory system prototypes and the relative performance of the three interface schemes. It is possible to comment on the performance of the interface schemes by using an unlimited-resource processor with a particular first-level cache prototype. The processor resource usage is used to derive prototypes by application of the techniques of the previous chapter. Comparison of this processor prototype to the those derived assuming no cache miss

penalty serves as a method of evauating the effects of processor/memory interaction on prototype selection.

This section uses members of the SPEC89 benchmark set as inputs to a combined processor/data cache simulation. The *blocking* scheme does not require special simulation at all (as is shown below). The other two schemes do, however, require special simulation. The simulation algorithm used for these schemes is a variant of the queue-based algorithm presented in Chapter 4. The modifications are not discussed in detail here. The changes involve the addition of a traditional cache simulator to the *window* queue scan. This simulator is invoked for every *Load/Store* instruction in the instruction trace. Instructions that cause a miss are dealt with in accordance to the scheme being simulated.

Chapters 3 and 4 discuss methods for simulation based on statistical sampling. For the uniprogramming case, sampling for traditional cache simulation is difficult to apply in a cycle-by-cycle simulation such as is used here. The difficulty is due to the state repair problem. The hit/miss status of each reference to the cache must be known exactly. Since the system is under a multiprogramming load, an average context switching interval is assumed. It was suggested by Laha et al. [5]. that the sample size be set equal to the average context switching interval. The cache is then assumed to be flushed between each sample of instructions. This approach is used here. The multiprogramming caches presented in Tables 3.6 and 3.7 of Chapter 3 are used to derive prototypes for the memory system for each benchmark. The memory systems are selected using the criteria $\rho \leq \hat{\rho}$, for $\hat{\rho} = 0.05$ and $\hat{\rho} = 0.10$.

5.2.1 Performance of *blocking* scheme

The performance of the *blocking* scheme can be re-created using a simple analytic model without the need for additional simulation. Recall that T_{MISS} was defined to be the number of cycles needed to repair a cache miss. From Chapter 3, N is the total number of addresses referenced. In Section 4.1.3, N_I was defined to be the total number of instructions executed. The additional cycles wasted due to blocking is $\rho \cdot (N\,T_{MISS})$. Therefore, the instruction per cycle metric for blocking is

$$\text{IPC}^* = \frac{N_I}{T_{TOT} + \rho \cdot (N\,T_{MISS})}. \tag{5.1}$$

It is clear that since $\rho = \rho^*$, the data cache prototype need not change when using the *blocking* scheme. As discussed in Section 5.1, since the clock is halted when a cache miss occurs, a cache miss has the same latency as a cache hit from the perspective of the processor. The resource usage of the processor and, therefore, the processor prototype, is not changed from the ideal case for the *blocking* scheme.

The performance of the unlimited-resource processors with the $\hat{\rho} = 0.05$ and $\hat{\rho} = 0.10$ data cache prototypes demonstrates the problems with the *blocking* schemes. Tables 5.1 presents the IPC* for the benchmarks for issue rates of two, four and eight instructions per cycle, data cache prototypes $\hat{\rho} = 0.05$ and 0.10, $T_{MISS} = 10$. Table 5.2 presents similar information for $T_{MISS} = 20$. The geometric mean of IPC* across all benchmarks is also presented in both tables. From the tables, the best mean performance is for $T_{MISS} = 10$, issue rate two, $\hat{\rho} = 0.05$, the least-demanding situation. Even in this case, the *blocking* scheme's performance is 79% of that for the perfect case of no cache misses

Table 5.1: IPC* (and percentage of *perfect* IPC) for *blocking* scheme, issue rates of two, four and eight instructions per cycle, data cache prototypes $\hat{\rho} = 0.05$ and 0.10, $T_{MISS} = 10$.

Bench-	issue rate = 2				issue rate = 4				issue rate = 8			
mark	$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$	
doduc	1.43	79%	1.21	67%	2.16	69%	1.69	54%	2.85	59%	2.10	44%
eqntott	1.37	73%	1.25	66%	2.06	59%	1.80	52%	2.75	46%	2.30	38%
espresso	1.21	76%	1.14	72%	1.63	68%	1.50	62%	1.90	62%	1.73	57%
gcc	1.17	79%	1.11	74%	1.50	73%	1.39	67%	1.69	70%	1.55	64%
matrix300	*		1.43	72%	*		2.22	56%	*		3.06	40%
xlisp	0.86	79%	0.83	76%	0.98	77%	0.94	73%	1.04	75%	1.00	72%
GMean	1.19	77%	1.10	71%	1.61	69%	1.43	61%	1.92	62%	1.67	54%

(* data cache design criteria could not be met)

Table 5.2: IPC* (and percentage of *perfect* IPC) for *blocking* scheme, issue rates of two, four and eight instructions per cycle, data cache prototypes $\hat{\rho} = 0.05$ and 0.10, $T_{MISS} = 20$.

Bench-	issue rate = 2				issue rate = 4				issue rate = 8			
mark	$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$	
doduc	1.19	66%	0.91	51%	1.65	53%	1.16	37%	2.02	42%	1.34	28%
eqntott	1.08	57%	0.94	50%	1.47	42%	1.21	35%	1.78	30%	1.42	24%
espresso	0.97	61%	0.88	55%	1.23	51%	1.09	45%	1.37	45%	1.20	39%
gcc	0.97	65%	0.88	59%	1.18	57%	1.05	51%	1.29	54%	1.14	47%
matrix300	*		1.12	56%	*		1.55	39%	*		1.92	25%
xlisp	0.71	65%	0.67	61%	0.79	62%	0.74	58%	0.83	60%	0.78	56%
GMean	0.97	63%	0.87	56%	1.23	53%	1.04	45%	1.39	45%	1.15	37%

(* data cache design criteria could not be met)

for some benchmarks (doduc, gcc and xlisp) and 77% on average. The performance degrades considerably at high issue rates: 38% for eqntott, issue rate eight, $T_{MISS} = 10$ and 24% for the same benchmark and issue rate when $T_{MISS} = 20$.

Clearly the *blocking* scheme has unacceptable performance. The impact of this observation is that although the *blocking* scheme has ideal processor/memory interaction, a desirable quality for combining prototyped subsystems, its performance is marginally acceptable for issue rates of two instructions per cycle and unacceptable for issue rates of four and eight instructions per cycle.

5.2.2 Performance of *limited-blocking* scheme

The performance results for the *limited-blocking* scheme used with a processor having unlimited resources, issue rates of two, four and eight instructions per cycle, data cache prototypes $\hat{\rho} = 0.05$ and 0.10, and $T_{MISS} = 10$ are presented in Table 5.3. Similar results for $T_{MISS} = 20$ are presented in Table 5.4. The *limited-blocking* scheme has acceptable performances for issue rates of two and four instructions per cycle when $T_{MISS} = 10$ (Table 5.3). For the $\hat{\rho} = 0.05$ data cache prototypes, the mean performance is 77% and 69% of the perfect case for issue rates two and four instructions per cycle, respectively. The performance is poor for an issue rate of eight instructions per cycle when $T_{MISS} = 10$. The performance for $T_{MISS} = 20$ is marginal for a low issue rate of two instructions per cycle (Table 5.4).

Table 5.3: IPC* (and percentage of *perfect* IPC) for *limited-blocking* scheme, issue rates of two, four and eight instructions per cycle, data cache prototypes $\hat{\rho} = 0.05$ and 0.10, $T_{MISS} = 10$.

Bench-	issue rate = 2				issue rate = 4				issue rate = 8			
mark	$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$	
doduc	1.67	93%	1.62	90%	2.58	83%	2.20	71%	3.37	70%	2.51	53%
eqntott	1.72	91%	1.59	85%	2.80	80%	2.35	68%	3.88	65%	3.02	50%
espresso	1.41	89%	1.36	86%	1.87	78%	1.76	73%	2.13	70%	1.95	64%
gcc	1.33	89%	1.29	87%	1.70	83%	1.61	78%	1.87	78%	1.74	72%
matrix300	*		1.94	97%	*		3.49	89%	*		4.05	53%
xlisp	0.93	85%	0.90	83%	1.02	80%	0.99	77%	1.08	78%	1.04	75%
GMean	1.38	90%	1.32	86%	1.88	81%	1.71	73%	2.24	72%	1.93	62%

(* data cache design criteria could not be met)

Table 5.4: IPC* (and percentage of *perfect* IPC) for *limited-blocking* scheme, issue rates of two, four and eight instructions per cycle, data cache prototypes $\hat{\rho} = 0.05$ and 0.10, $T_{MISS} = 20$.

Bench-	issue rate = 2				issue rate = 4				issue rate = 8			
mark	$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$	
doduc	1.44	80%	1.21	67%	1.95	63%	1.39	45%	2.31	48%	1.49	31%
eqntott	1.48	79%	1.23	65%	2.06	59%	1.58	45%	2.30	38%	1.73	29%
espresso	1.16	73%	1.07	67%	1.40	58%	1.24	51%	1.51	50%	1.32	43%
gcc	1.13	76%	1.05	70%	1.35	34%	1.21	31%	1.43	59%	1.27	53%
matrix300	*		1.76	88%	*		2.05	52%	*		2.05	27%
xlisp	0.76	70%	0.72	66%	0.82	64%	0.77	60%	0.85	61%	0.80	58%
GMean	1.16	75%	1.04	68%	1.44	62%	1.20	52%	1.58	51%	1.28	41%

(* data cache design criteria could not be met)

In general, the performance of *limited-blocking* for $T_{MISS} = 10$ is sufficient to warrant its use for situations where $T_{MISS} = 10$ and the issue rate is either two or four instructions per cycle.

Unlike the *blocking* scheme, the *limited-blocking* scheme alters the processor resource usage in the presence of cache misses. This effect can alter the assumptions used for processor prototype selection. Let P_z be the processor prototype selected assuming perfect processor/memory interaction (i.e., *perfect cache*) for some criterion, z. Figures 5.3 ($T_{MISS} = 10$) and 5.4 ($T_{MISS} = 20$) present the geometric mean of the IPC metric for the prototypes of Chapter 4, $z = 0.99$, 0.95, and 0.80, when the prototypes are interconnected with memory using the *limited-blocking* scheme. Also present in the figures are the IPC values for the unlimited resource cases from Tables 5.3 and 5.4. The figures demonstrate that prototype performance continues to be a function of prototype selection criteria after the processor prototype is connected to the memory system. Comparison of the figures to Figure 4.10 of Chapter 4 (page 120) reveals that the difference between prototype performance for differing values of z is less dramatic in the presence of cache misses using the *limited-blocking* scheme than it is in the case of perfect cache performance. Increasing the value of z translates into increased performance, yet this additional performance is now much less evident.

It is possible to use the processor resource usage of the simulations of the *limited-blocking* scheme to select new processor prototypes. Table 5.5 presents the difference

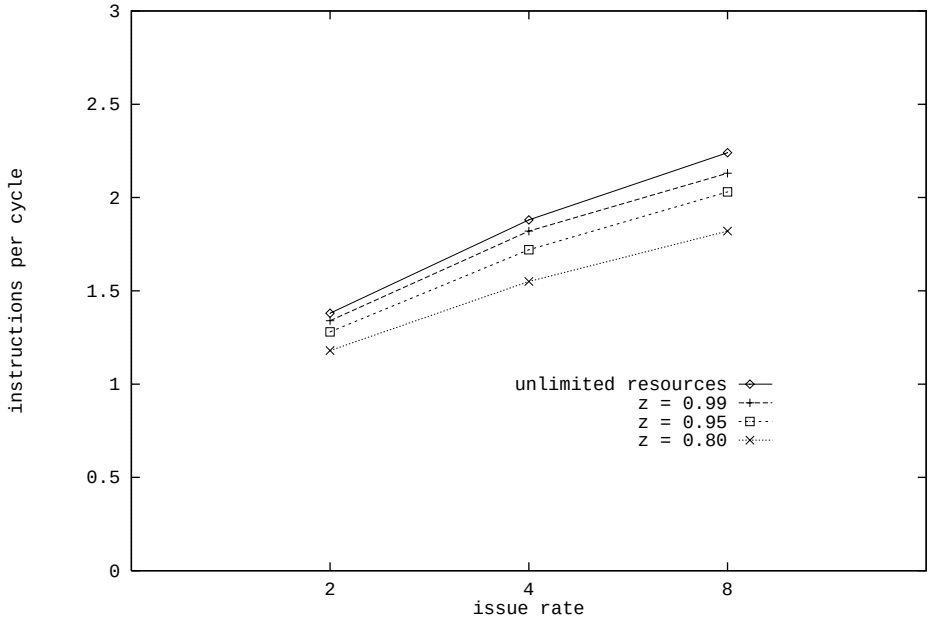

Figure 5.3: Geometric mean of IPC for prototypes $z = 0.99$, 0.95, and 0.80 compared to unlimited-resource IPC for *limited-blocking* scheme, $\hat{\rho} = 0.05$, $T_{MISS} = 10$.

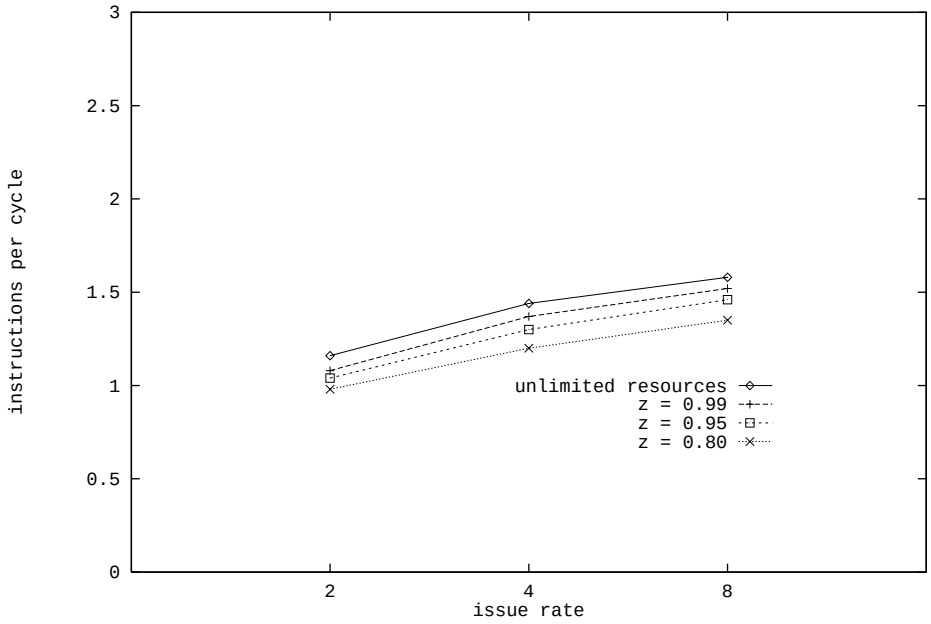

Figure 5.4: Geometric mean of IPC for prototypes $z = 0.99$, 0.95, and 0.80 compared to unlimited-resource IPC for *limited-blocking* scheme, $\hat{\rho} = 0.05$, $T_{MISS} = 20$.

between the prototype designs of the *perfect cache* and this *limited-blocking* scheme processor prototype for $z = 0.99$, $T_{MISS} = 10$, $\hat{\rho} = 0.05$, and an issue rate of two instructions per cycle. Negative values in the table indicate that the *limited-blocking* scheme prototype requires less resources than the original (*perfect cache*) prototype. Negative values are favorable, indicating that the original prototype has sufficient resources to satisfy the benchmark requirements after the processor and memory system are interconnected. Zero or negative values occur in Table 5.5 for all benchmarks except for xlisp that requires one additional *IAlu* function unit when interconnected. This result is much stronger for determining processor/memory interaction than the comparison of the performance of Figures 5.3 and 5.4 to Figure 4.10 of Chapter 4. It indicates that the techniques of Chapter 4 are valid for this particular set of parameters after the prototype is used in the presence of real cache misses.

The interaction between processor and memory is sufficient to invalidate the prototypes selected in Chapter 4 when the performance of the interconnection scheme is low. Consider the case of an issue rate of four instructions per cycle, $T_{MISS} = 10$, that has the marginally acceptable performance shown in Table 5.3. Table 5.6 presents the difference between the prototype designs of the *perfect cache* and the *limited-blocking* scheme processor prototype for $z = 0.99$, $T_{MISS} = 10$, $\hat{\rho} = 0.05$ for an issue rate of four instructions per cycle. Positive values in this table indicate that the original processor prototype did not supply sufficient resources. These occur for all benchmarks in the case of *Load* function units. The range of differences in the function unit number of the prototype designs

is presented graphically for all three issue rates in Figure 5.5 for the same parameters as Tables 5.5 and 5.6 ($z = 0.99$, $T_{MISS} = 10$, and $\hat{\rho} = 0.05$). Notice that when the issue rate is increased to eight instructions per cycle, the extra number of needed function units can be as high as nine for some benchmarks. These results suggest that the prototype methods are not universally valid when the *limited-blocking* scheme is used.

Table 5.5: Difference between prototype designs of *perfect cache* and *limited-blocking* scheme, $z = 0.99$, $T_{MISS} = 10$, $\hat{\rho} = 0.05$, issue rate of two instructions per cycle.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	-1	0	0	-2	-1	-1	-1
eqntott	-1	0	0	-1	0	-1	-2
espresso	-3	0	0	-2	0	-2	0
gcc	-2	0	0	-2	-1	-1	-1
xlisp	1	0	0	-1	0	0	0
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	-2	-1	-2	0	-1	-1	0
eqntott	0	0	0	0	0	-1	-1
espresso	-1	0	0	0	0	-1	-1
gcc	-2	0	0	0	0	0	-1
xlisp	0	0	0	0	0	0	0

Table 5.6: Difference between prototype designs of *perfect cache* and *limited-blocking* scheme, $z = 0.99$, $T_{MISS} = 10$, $\hat{\rho} = 0.05$, issue rate of four instructions per cycle.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	-1	0	0	-2	-1	0	3
eqntott	3	0	0	0	0	-1	5
espresso	-2	0	0	-2	0	-1	4
gcc	0	0	0	-1	0	-1	1
xlisp	1	0	0	-1	0	0	1
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	-1	0	0	0	0	-1	0
eqntott	0	0	0	0	0	2	2
espresso	-1	0	0	0	0	-1	-1
gcc	-2	0	0	0	0	0	-1
xlisp	0	0	0	0	0	0	0

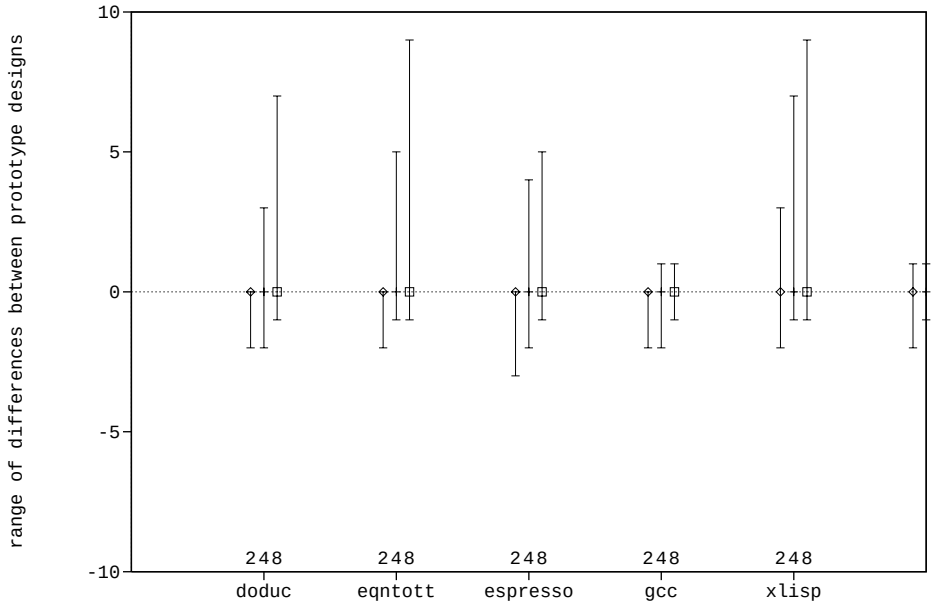

Figure 5.5: Range of differences between prototypes for *limited-blocking* scheme, $z = 0.99$, $T_{MISS} = 10$, $\hat{\rho} = 0.05$, issue rates of two, four and eight instructions per cycle.

5.2.3 Performance of *non-blocking*

The *non-blocking* scheme for addressing processor/memory interaction is the most-complicated of the three schemes. This complex hardware is justified by the increased performance that the *non-blocking* scheme provides. This performance is shown by simulating unlimited processor resources for each benchmark with issue rates of two, four, and eight instructions per cycle. The results are presented for $T_{MISS} = 10$ and $T_{MISS} = 20$ in Tables 5.7 and 5.8, respectively. The tables demonstrate that the *non-blocking* scheme can achieve approximately 79%–93% of the no cache miss performance with unlimited resources. The worst-case performance of 79% occurs in the most-demanding case of $\hat{\rho} = 0.10$, an issue rate of eight instructions per cycle and a cache miss penalty of $T_{MISS} = 20$. These results suggest that this scheme is required for systems that issue many instructions per cycle or have a high miss penalty.

The processor resource usage of the simulations of the *non-blocking* scheme is used to select new processor prototypes. The number of entries in the pending miss queue is also prototyped using the methods of Chapter 4. Figures 5.6 ($T_{MISS} = 10$) and 5.7 ($T_{MISS} = 20$) present the geometric mean of the IPC for the prototypes of Chapter 4 for values of $z = 0.99$, 0.95, and 0.80. Also present in the figures are the IPC values for the unlimited resource case from Tables 5.7 and 5.8. These can be compared with Figure 4.10 of Chapter 4 to reveal similar relative differences between prototype performance across values of z. Stronger results are presented in Tables 5.9–5.11 for values of $z = 0.99$, $T_{MISS} = 10$ and 20, and $\hat{\rho} = 0.05$ and 0.10. These four tables are for the most-demanding

Table 5.7: IPC* (and percentage of *perfect* IPC) for *non-blocking* scheme, issue rates of two, four and eight instructions per cycle, data cache prototypes $\hat{\rho} = 0.05$ and 0.10, $T_{MISS} = 10$.

Bench-	issue rate = 2				issue rate = 4				issue rate = 8			
mark	$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$	
doduc	1.77	98%	1.74	97%	3.02	97%	2.96	95%	4.59	95%	4.43	92%
eqntott	1.87	99%	1.86	99%	3.41	98%	3.39	97%	5.80	97%	5.74	96%
espresso	1.50	94%	1.46	92%	2.19	91%	2.12	88%	2.66	87%	2.55	84%
gcc	1.39	93%	1.36	91%	1.86	90%	1.81	88%	2.14	89%	2.07	86%
matrix300	*		1.98	99%	*		3.89	99%	*		7.47	98%
xlisp	0.95	87%	0.94	86%	1.09	85%	1.07	84%	1.17	84%	1.15	83%
GMean	1.46	95%	1.43	93%	2.15	92%	2.10	90%	2.82	90%	2.74	88%

(* data cache design criteria could not be met)

Table 5.8: IPC* (and percentage of *perfect* IPC) for *non-blocking* scheme, issue rates of two, four and eight instructions per cycle, data cache prototypes $\hat{\rho} = 0.05$ and 0.10, $T_{MISS} = 20$.

Bench-	issue rate = 2				issue rate = 4				issue rate = 8			
mark	$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$		$\hat{\rho} = 0.05$		$\hat{\rho} = 0.10$	
doduc	1.74	97%	1.70	94%	2.93	94%	2.82	90%	4.39	91%	4.13	86%
eqntott	1.85	98%	1.83	97%	3.35	96%	3.31	95%	5.62	94%	5.53	92%
espresso	1.41	89%	1.35	85%	1.99	83%	1.87	78%	2.37	78%	2.20	72%
gcc	1.30	87%	1.26	85%	1.71	83%	1.63	79%	1.94	80%	1.83	76%
matrix300	*		1.97	99%	*		3.86	98%	*		7.35	97%
xlisp	0.85	78%	0.83	76%	0.97	76%	0.94	73%	1.03	74%	1.00	72%
GMean	1.38	90%	1.34	87%	2.01	86%	1.93	83%	2.59	83%	2.47	79%

(* data cache design criteria could not be met)

issue rate of eight instructions per cycle. The range of these results is also presented graphically for $z = 0.99$ and $\hat{\rho} = 0.05$ in Figures 5.8 ($T_{MISS} = 10$) and 5.9 ($T_{MISS} = 10$). The tables and figures display little or no differences between the prototypes, with the exception of the eqntott benchmark that requires two additional *IAlu* and *AddrC* function units and five additional *Load* function units. With this exception, the results validate that the *non-blocking* scheme has very low processor/memory interaction and very high performance.

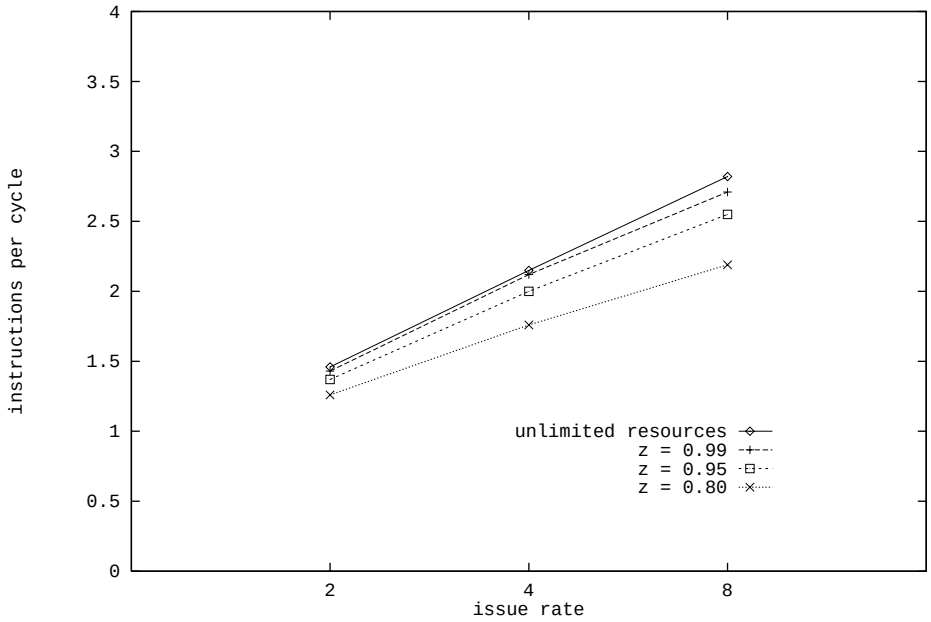

Figure 5.6: Geometric mean of IPC for prototypes $z = 0.99$, 0.95, and 0.80 compared to unlimited-resource IPC for *non-blocking* scheme, $\hat{\rho} = 0.05$, $T_{MISS} = 10$.

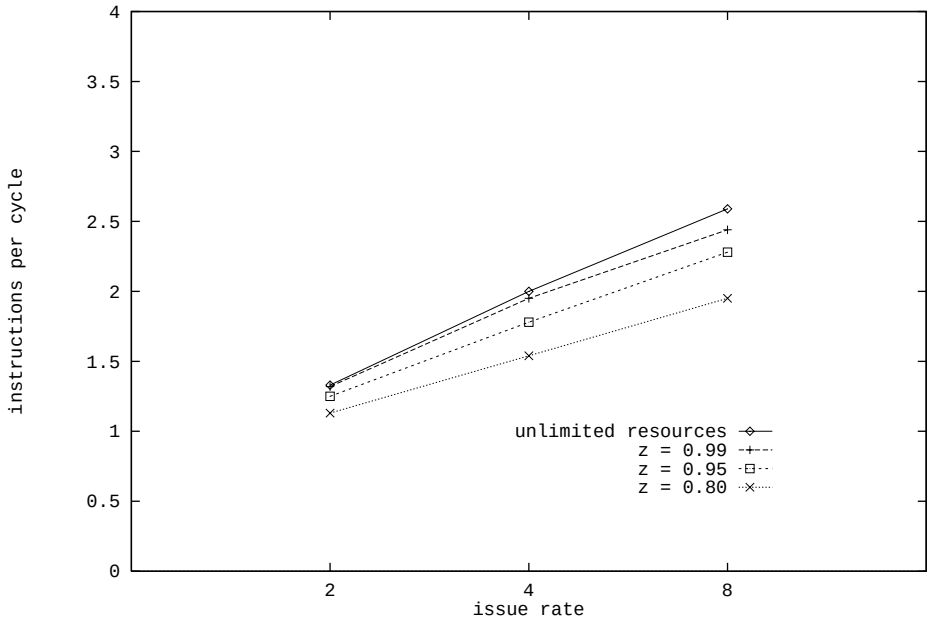

Figure 5.7: Geometric mean of IPC for prototypes $z = 0.99$, 0.95, and 0.80 compared to unlimited-resource IPC for *non-blocking* scheme, $\hat{\rho} = 0.05$, $T_{MISS} = 20$.

Table 5.9: Difference between prototype designs of *perfect cache* and *non-blocking* scheme, $z = 0.99$, $T_{MISS} = 10$, $\hat{\rho} = 0.05$, issue rate of eight instructions per cycle.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	0	0	0	0	0	0	1
eqntott	2	0	0	2	0	0	5
espresso	-1	0	0	0	0	0	1
gcc	0	0	0	0	0	0	0
xlisp	1	0	0	0	0	0	0
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	0	1	1	0	0	-1	0
eqntott	1	0	0	0	0	0	0
espresso	0	0	0	0	0	-1	0
gcc	0	0	0	0	0	0	0
xlisp	0	0	0	0	0	0	0

Table 5.10: Difference between prototype designs of *perfect cache* and *non-blocking* scheme, $z = 0.99$, $T_{MISS} = 20$, $\hat{\rho} = 0.05$, issue rate of eight instructions per cycle.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	0	0	0	0	0	0	1
eqntott	2	0	0	2	0	0	5
espresso	-1	0	0	0	0	0	0
gcc	0	0	0	0	0	0	0
xlisp	0	0	0	0	0	0	0
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	0	1	1	0	0	-1	0
eqntott	0	0	0	0	0	0	0
espresso	0	0	0	0	0	-1	-1
gcc	-1	0	0	0	0	0	-1
xlisp	0	0	0	0	0	0	0

Table 5.11: Difference between prototype designs of *perfect cache* and *non-blocking* scheme, $z = 0.99$, $T_{MISS} = 10$, $\hat{\rho} = 0.10$, issue rate of eight instructions per cycle.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	0	0	0	0	0	0	1
eqntott	2	0	0	2	0	0	5
espresso	-1	0	0	0	0	0	1
gcc	0	0	0	0	0	0	0
xlisp	1	0	0	0	0	0	0
matrix300	0	1	0	0	0	0	0
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	0	1	1	0	0	-1	0
eqntott	0	0	0	0	0	0	0
espresso	0	0	0	0	0	-1	0
gcc	0	0	0	0	0	0	0
xlisp	0	0	0	0	0	0	0
matrix300	0	0	0	0	0	-1	0

Table 5.12: Difference between prototype designs of *perfect cache* and *non-blocking* scheme, $z = 0.99$, $T_{MISS} = 20$, $\hat{\rho} = 0.10$, issue rate of eight instructions per cycle.

Benchmark	IAlu	IMul	IDiv	AddrC	Move	Shift	Load
doduc	0	0	0	0	0	0	1
eqntott	2	0	0	2	0	0	5
espresso	-1	0	0	-1	0	-1	0
gcc	0	0	0	0	0	0	0
xlisp	0	0	0	0	0	0	0
matrix300	0	1	0	0	0	0	0
	Store	FPAdd	FPMul	FPDiv	FPCvt	Branch	Test
doduc	0	1	1	0	0	-1	0
eqntott	0	0	0	0	0	0	0
espresso	-1	0	0	0	0	-1	-1
gcc	-1	0	0	0	0	0	-1
xlisp	0	0	0	0	0	0	0
matrix300	0	0	0	0	0	-1	0

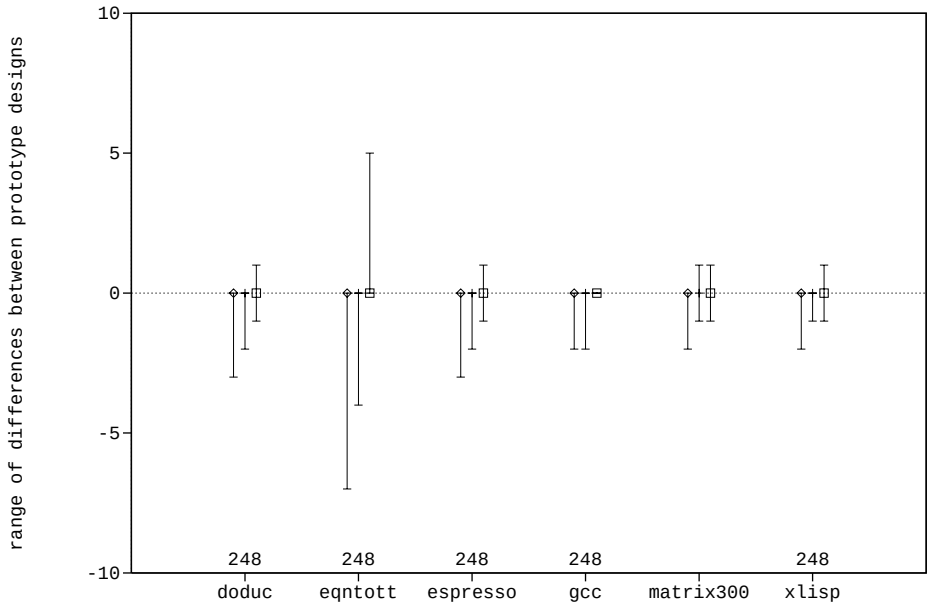

Figure 5.8: Range of differences between prototypes for *non-blocking* scheme, $z = 0.99$, $T_{MISS} = 10$, $\hat{\rho} = 0.10$, and issue rates of two, four and eight instructions per cycle.

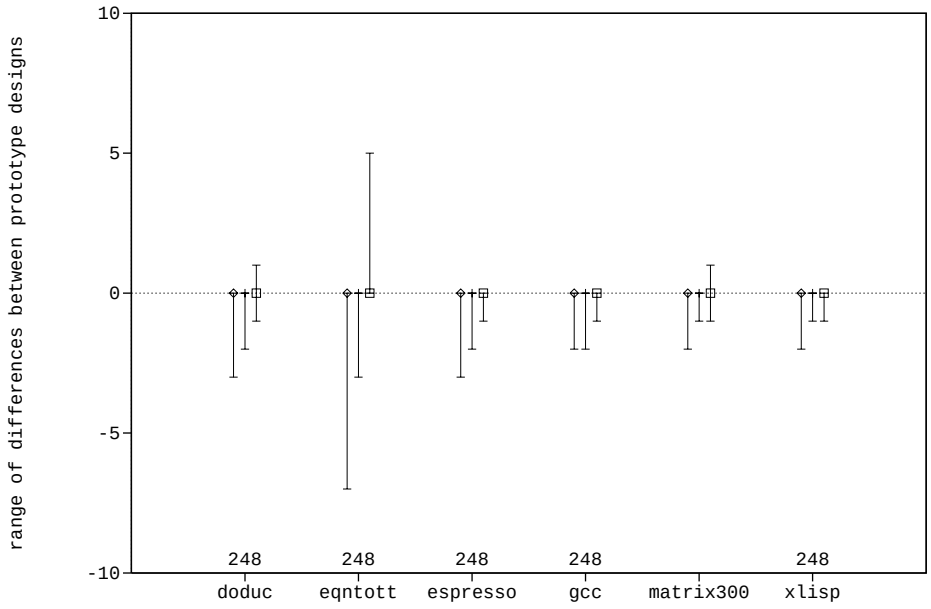

Figure 5.9: Range of differences between prototypes for *non-blocking* scheme, $z = 0.99$, $T_{MISS} = 20$, $\hat{\rho} = 0.10$, issue rates of two, four and eight instructions per cycle.

5.3 Summary of Processor/Memory Interaction

Earlier chapters of this thesis assumed that the processor and memory components could be designed independently and then combined without the need for re-design. This chapter presented and evaluated three possible architectural schemes for interconnecting the processor and the memory system: *blocking*, *limited-blocking*, and *non-blocking*. The *blocking* scheme had perfect processor/memory interaction but low performance. Of the other two schemes, when the schemes possessed high performance, the processor/memory interaction did not alter the $z = 0.99$ processor prototypes.

The *blocking* scheme is appealing because the processor state is unchanged by a cache miss. It was found that the scheme had the worst performance of the three schemes, however. It is suggested that this scheme only be used in designs that have $\hat{\rho} \leq 5\%$, $T_{MISS} \leq 10$ cycles and issue rates of two instructions per cycle.

The *limited-blocking* scheme was evaluated and found to have high performance in some situations, specifically, when $T_{MISS} = 10$ cycles and issue rates of two or four instructions per cycle. The interaction for the low issue rate of two instructions per cycle cases was not sufficient to warrant changes in processor prototype design. This was no longer true for the issue rate of four instructions per cycle where the processor prototypes of Chapter 4 became invalid.

The highest-performance scheme was found to be the *non-blocking* scheme, having IPC close to the ideal (i.e., no cache-miss) performance. This scheme also hid processor/memory interaction well. The $z = 0.99$ processor prototypes performed nearly as

well as the unlimited-resource case for all benchmarks. Additionally, the prototypes derived from the *non-blocking* simulations were nearly identical to the original prototypes derived assuming perfect cache performance.

The geometric means of IPC* across all benchmarks for the three schemes and the *perfect* case (no cache miss penalty) are shown in Figures 5.10 ($T_{MISS} = 10$) and 5.11 ($T_{MISS} = 20$) for the $\hat{\rho} = 0.10$ data cache prototypes. These two graphs show the relative

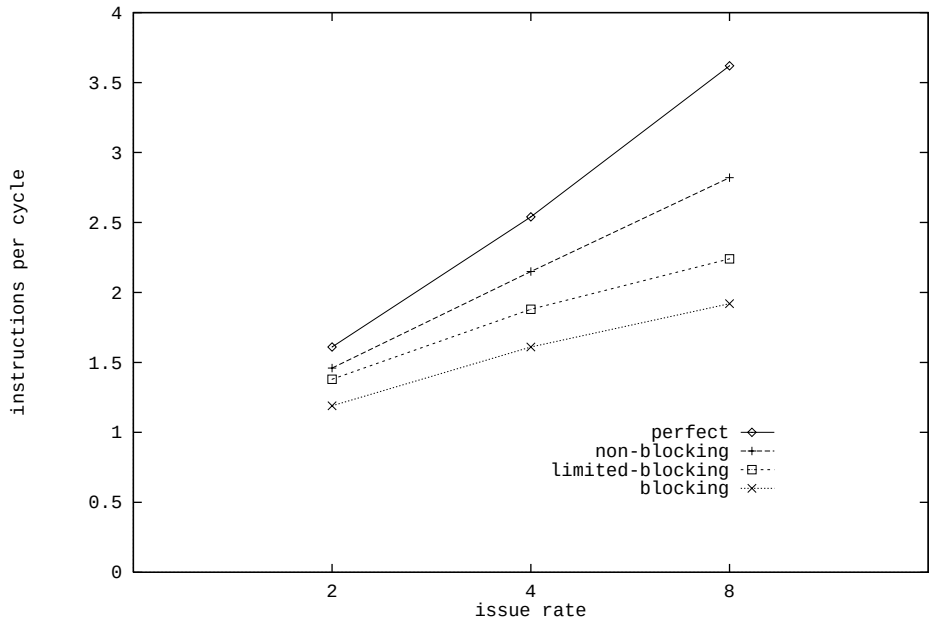

Figure 5.10: The geometric mean of IPC* and IPC for *perfect* case for issue rates two, four and eight instructions per cycle, $T_{MISS} = 10$.

ordering between the schemes from low to high performance as: *blocking*, *limited-blocking*, and *non-blocking*. The data demonstrate that sufficient design options exist to justify designing the processor and the memory separately and then combining them without redesign.

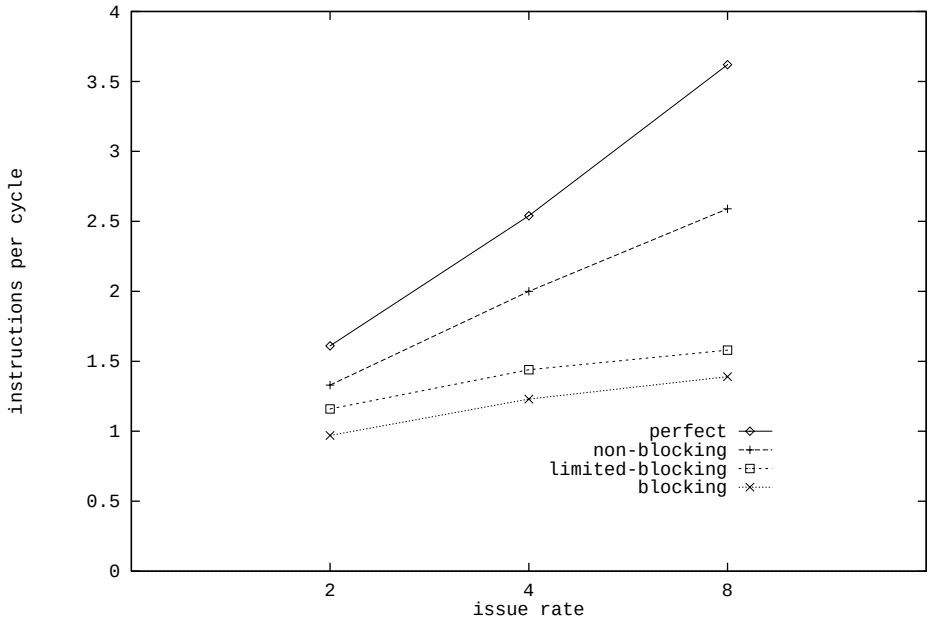

Figure 5.11: The geometric mean of IPC* and IPC for *perfect* case for issue rates two, four and eight instructions per cycle, $T_{MISS} = 20$.

6. CONCLUSIONS

This thesis has addressed the problem of improving the design of modern computers by improvement of the design process. The current design process is over-taxed due to the size and diversity of realistic workload models. A method was suggested to improve the design process by deriving prototype systems from workloads. This idea is similar in philosophy to *synthesis* techniques from the field of computer-aided design. What makes this thesis novel is that it has developed a set of techniques to perform synthesis at an architectural level. This thesis has concentrated on the processor and memory components of workstations, although the overall philosophy is applicable to other components as well.

Techniques developed for prototyping cache-based memory hierarchies centered around developing fast and accurate techniques to evaluate a large space of possible cache designs. Single-pass cache simulation methods were suggested as the starting point for these techniques and the recurrence/conflict version of the single-pass method was derived. The speed of this technique was improved via adaptation of statistical sampling

from traditional cache simulation methods. Two such adaptations were suggested and compared. The *fill-flush* technique was evaluated yet found to have low accuracy. To correct this problem, a new technique that exploits the properties of single-pass methods was derived. This technique was found to produce results of higher accuracy at the expense of some additional simulation time.

Multiprogramming effects on cache design selection were addressed by developing a model of involuntary context switching effects that is independent of the intensity of context switching. A single-pass method based on the model that also includes the measurement of voluntary context switch events (i.e., system calls) was developed and its behavior was shown to be consistent with the model.

The single-pass techniques were combined in a process used to select prototypes from the resultant space of designs. This unified approach was demonstrated with the exploration of 324 potential data cache designs for the uniprogramming case and an additional 324 designs under the assumption of a multiprogramming workload.

The processor component of a workstation was discussed and decomposed using a five-stage model. A classification was suggested inspired by Flynn notation, and the member of the classification with the highest potential performance, FOCO-class processors, was discussed in detail. Two simulation algorithms were derived for the FOCO-class processors. One method is based on three queues that correspond closely to the actual hardware design. The second method is event driven and derives its results directly from the instruction trace.

Statistical sampling was further adapted from traditional cache simulation and applied to the processor simulation algorithms. This technique was found to be highly accurate with relative errors of approximately 10% for the IPC metric when using 40 samples of 10,000 instructions each.

Two methods were proposed to derive prototypes of the processor. One method keeps constant parameters such as instruction issue rate and function unit latency but leaves the number of function units unlimited during simulation. Prototype selection criteria based on the cumulative resource usage distribution function were evaluated empirically. It was found that these criteria produce prototypes having performance that is bounded by the ideal performance, confirming their usefulness for processor prototyping.

The case in which the function unit latencies are variable was investigated and formulated as a general optimization problem. A version of *simulated annealing* was applied to this problem and several example annealing schedules were shown to demonstrate the technique.

The interaction between prototypes was investigated via simulation by studying three schemes for coupling the processor and memory components together. One scheme, *blocking*, has the desirable quality of not requiring alteration of the prototype designs. However, the *blocking* scheme also had the lowest performance for FOCO-class processors. Two additional techniques were evaluated, *limited-blocking* and *non-blocking*. The *limited-blocking* scheme was found to perform near the ideal for previously selected prototypes for issue rates of two and four instructions per cycle and a miss penalty of 10

cycles. Similar results were found for the *non-blocking* scheme, which also had the highest performance of the three schemes. In general, it was concluded that sufficient design options existed to justify designing the processor and the memory separately and then combining them without redesign.

REFERENCES

[1] N. Cross, *Engineering Design Methods*. New York: John Wiley & Sons, 1989.

[2] G. Cybenko, L. Kipp, L. Pointer, and D. Kuck, "Supercomputer performance evaulation and the Perfect Club," in *Proc. Int'l. Conf. on Supercomputing*, (Amsterdam, The Netherlands), pp. 254–266, June 1990.

[3] "Spec newsletter," Feb. 1989. SPEC, Fremont, CA.

[4] R. L. Mattson, J. Gercsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, 1970.

[5] S. Laha, J. A. Patel, and R. K. Iyer, "Accurate low-cost methods for performance evaluation of cache memory systems," *IEEE Trans. Comput.*, vol. C-37, pp. 1325–1336, Feb. 1988.

[6] T. M. Conte and W. W. Hwu, "Benchmark characterization," *IEEE Computer*, pp. 48–56, Jan. 1991.

[7] B. Kumar and E. S. Davidson, "Computer system design using a hierarchial approach to performance evaluation," *Communications ACM*, vol. 23, pp. 511–521, Sept. 1980.

[8] C. G. Ponder, "An analytical look at linear performance models," Tech. Rep. UCRL-JC-106105, Lawrence Livermore National Laboratory, Livermore, CA, Sept. 1990.

[9] R. H. Saavedra-Barrera, "Machine characterization and benchmark performance prediction," Tech. Rep. UCB/CSD 88/437, Computer Science Division, University of California, Berkeley, CA, June 1988.

[10] R. H. Saavedra-Barrera, A. J. Smith, and E. Miya, "Machine characterization based on an abstract high-level language machine," *IEEE Trans. Comput.*, vol. 38, pp. 1659–1679, Dec. 1989.

[11] J. W. C. Fu and J. H. Patel, "How to simulate 100 billion references cheaply," Tech. Rep. CRHC-91–30, Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, IL, Nov. 1991.

[12] D. W. Clark, "Cache performance in the VAX-11/780," *ACM Trans. Computer Systems*, vol. 1, pp. 24–37, Feb. 1983.

[13] K. R. Kaplan and R. O. Winder, "Cache-based computer systems," *Computer*, vol. 6, pp. 30–36, Mar. 1973.

[14] W. D. Strecker, "Cache memories for PDP-11 family computers," in *Proc. 3rd Ann. Int'l. Symp. Computer Architecture*, pp. 155–158, Jan. 1976.

[15] G. S. Shedler and D. R. Slutz, "Derivation of miss ratios for merged access streams," *IBM Journal of Research and Development*, vol. 20, pp. 505–517, Sept. 1976.

[16] M. C. Easton, "Computation of cold-start miss ratios," *IEEE Trans. Comput.*, vol. C-27, pp. 404–408, May 1978.

[17] A. J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[18] I. J. Haikala, "Cache hit ratios with geometric task switch intervals," in *Proc. 11th Ann. Int'l. Symp. Computer Architecture*, (Ann Arbor, MI), pp. 364–371, June 1984.

[19] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache performance of operating system and multiprogramming workloads," *ACM Transactions on Computer Systems*, vol. 6, pp. 393–431, Nov. 1988.

[20] T. M. Conte and W. W. Hwu, "A brief survey of benchmark usage in the architecture community," *Computer Architecture News (ACM SIGARCH)*, May 1991.

[21] J. R. Larus, "Abstract execution: a technique for efficiently tracing programs," Tech. Rep., Computer Sciences Department, University of Wisconsin–Madison, Feb. 1990.

[22] C. B. Stunkel and W. K. Fuchs, "TRAPEDS: producing traces for multicomputers via execution driven simulation," in *Proc. ACM SIGMETRICS '89 and PERFORMANCE '89 Int'l. Conf. on Measurement and Modeling of Computer Systems*, (Berkeley, CA), pp. 70–78, May 1989.

[23] M. L. Golden, "Issues in trace collection through program instrumentation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana-Champaign, IL, 1991.

[24] G. Kane, *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

[25] R. M. Stallman, *Using and porting GNU CC*. Free Software Foundation, Inc., 1989.

[26] D. W. Wall and M. L. Powell, "The Mahler experience: using an intermediate language as the machine description," in *Proc. Second Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, (Palo Alto, CA), pp. 100–104, Oct. 1987.

[27] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD Unix Operating System.* New York, NY: Addison-Wesley, 1989.

[28] A. J. Smith, "A second bibliography on cache memories," *Comput. Architecture News*, vol. 19, pp. 138–153, June 1991.

[29] A. Agarwal, M. Horowitz, and J. Hennessy, "An analytical cache model," *ACM Trans. Computer Systems*, vol. 7, pp. 184–215, May 1989.

[30] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Trans. Comput.*, vol. C-38, pp. 1612–1630, Dec. 1989.

[31] I. L. Traiger and D. R. Slutz, "One-pass techniques for the evaluation of memory hierarchies," IBM Research Report RJ 892, IBM, San Jose, CA, July 1971.

[32] J. G. Thompson, *Efficient analysis of caching systems*, Ph.D. dissertation, Computer Science Division, University of California, Berkeley, CA, Oct. 1987. Report No. UCB/CSD 87/374.

[33] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA: McGraw-Hill (MIT Press), 1990.

[34] D. A. Wood, M. D. Hill, and R. E. Kessler, "A model for estimating trace-sample miss ratios," in *Proc. ACM SIGMETRICS '91 Conf. on Measurement and Modeling of Comput. Sys.*, pp. 79–89, May 1991.

[35] H. S. Stone, *High-performance Computer Architecture.* New York, NY: Addison-Wesley, 1990.

[36] J. Emer and D. Clark, "A characterization of processor performance in the VAX-11/780," in *Proc. 11th Ann. Int'l. Symp. Computer Architecture*, (Ann Arbor, MI), p. 84–93, June 1984.

[37] D. W. Clark, P. J. Bannon, and J. B. Keller, "Measuring VAX 8800 performance with a histogram hardware monitor," in *Proc. 15th Ann. Int'l. Symp. Computer Architecture*, pp. 176–185, May 1988.

[38] D. Gannon, W. Jalby, and K. Gallivan, "Strategies for cache and local memory management by global program transformation," *J. Parallel and Distributed Computing*, vol. 5, pp. 587–616, 1988.

[39] J. A. Fisher and B. R. Rau, "Instruction-level parallel processing," *Science*, vol. 253, pp. 1233–1241, Sept. 1991.

[40] U. Banerjee, *Speedup of ordinary programs*, Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana-Champaign, IL, 1979.

[41] R. M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, Jan. 1967.

[42] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. C-21, pp. 948–960, Sept. 1972.

[43] J. E. Thornton, "Parallel operation in the Control Data 6600," in *Proc. AFIPS FJCC*, pp. 33–40, 1964.

[44] N. P. Jouppi and D. W. Wall, "Available instruction-level parallelism for superscalar and superpipelined machines," in *Proc. Third Int'l. Conf. on Architectural Support for Prog. Lang. and Operating Systems.*, pp. 272–282, Apr. 1989.

[45] S. Weiss and J. E. Smith, "Instruction issue logic for pipelined supercomputers," *IEEE Trans. Comput.*, vol. C-33, pp. 1013–1022, Nov. 1984.

[46] J. E. Smith and A. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *Proc. 12th Ann. Int'l. Symp. Computer Architecture*, (Boston, MA), June 1985.

[47] W. M. Johnson, *Super-scalar processor design.* Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA, June 1989.

[48] W. W. Hwu and Y. N. Patt, "Checkpoint repair for high performance out-of-order execution machines," *IEEE Trans. Comput.*, vol. C-36, p. 1496–1514, Dec. 1987.

[49] W. W. Hwu, T. M. Conte, and P. P. Chang, "Comparing software and hardware schemes for reducing the cost of branches," in *Proc. 16th Ann. Int'l. Symp. Computer Architecture*, (Jerusalem, Israel), pp. 224–233, June 1989.

[50] J. K. F. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, Jan. 1984.

[51] J. E. Smith, "A study of branch predition strategies," in *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, pp. 135–148, June 1981.

[52] M. Butler, T.-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow, "Single instruction stream parallelism is greater than two," in *Proc. 18th Ann. Int'l. Symp. Computer Architecture*, (Toronto, Canada), pp. 276–286, May 1991.

[53] J. H. Patel, personal communication, June 1992.

[54] G. S. Tjaden and M. J. Flynn, "Detection and parallel execution of independent instructions," *IEEE Trans. Comput.*, vol. C-19, pp. 889–895, Oct. 1970.

[55] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Evaluations of state calculations by fast computing machines," *J. Chem. Phys.*, vol. 21, pp. 1087–1092, 1953.

[56] S. Kirkpatric, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, May 1983.

[57] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vettering, *Numerical Recipies in C: The Art of Scientific Computing.* New York: Cambridge University Press, 1990.

[58] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. 8th Ann. Int'l. Symp. Computer Architecture*, pp. 81–87, May 1981.

VITA

Thomas Martin Conte was born on July 29, 1964, in Wilmington, Delaware. In 1984 he was selected as an Engineering Scholar while pursuing the degree of Bachelor of Electrical Engineering at the University of Delaware. In 1988, he received the M.S. degree in Electrical Engineering from the University of Illinois, Urbana-Champaign. From 1987 to 1988, he was a research assistant in the Center for Supercomputing Research and Development, the University of Illinois. From 1988 to 1992, he was a research assistant in the Center for Reliable and High-Performance Computing, also at the University of Illinois. In September, 1992 he will join the faculty of the Department of Electrical and Computer Engineering, the University of South Carolina, Columbia, South Carolina, as an assistant professor.