1993

Trace Driven Simulation of Cache Memories

Sridhar Adina
University of Rhode Island

Follow this and additional works at: <https://digitalcommons.uri.edu/theses>

Recommended Citation

Adina, Sridhar, "Trace Driven Simulation of Cache Memories" (1993). *Open Access Master's Theses*. Paper 872.
<https://digitalcommons.uri.edu/theses/872>

This Thesis is brought to you for free and open access by DigitalCommons@URI. It has been accepted for inclusion in Open Access Master's Theses by an authorized administrator of DigitalCommons@URI. For more information, please contact digitalcommons-group@uri.edu.

TRACE DRIVEN SIMULATION OF CACHE MEMORIES

BY

SRIDHAR ADINA

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

1993

MASTER OF SCIENCE THESIS
OF
Sridhar Adina

APPROVED:

Thesis Committee
Major Professor

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND
1993

Abstract

This thesis evaluates an innovative cache design called, prime-mapped cache. The performance analysis on various applications and programs shows that the prime-mapped cache performs better than the conventional cache organizations. The performance gain will increase with the increase of the speed gap between processors and memories. The exact cache behavior of numerical applications namely: matrix multiplication and SPEC benchmarks is studied by varying the cache parameters such as cachesize, linesize and associativity. Traces are collected from these programs and miss ratios for instructions and data accesses are compared. Based on the experimental results and depending on the algorithm used, the miss ratios of the prime-mapped cache are found to be 50 to 100% less than for conventional caches. Depending upon the speed difference between processors and memories, with the prime-mapped cache these algorithms can run 30% to 2 times faster than they do on conventional caches.

Acknowledgments

I would like to thank my major Professor Dr. Qing Yang for his guidance and cooperation in making this study a success and also providing financial support during part of the study.

I am also grateful to the faculty/staff members and fellow students in the Department of Electrical Engineering, who never hesitated to offer timely assistance and warm friendship. Particularly, I am grateful to all members of my thesis committee, Professor Daly and Professor Wolfe for their precious time and efforts. I would like to thank Professor Suryanarayan for being the chairman of the committee.

Contents

Abstract	ii
Acknowledgments	iii
Table of Contents	iv
List of Figures	vi
1 Introduction	1
1.1 Performance Analysis	1
1.2 Analytical Modeling	1
1.3 Measurement	2
1.4 Simulation	2
1.4.1 Event-Driven Simulation	2
1.4.2 Trace-Driven Simulation	2
1.5 Outline	4
2 Architecture Study	5
2.1 Background	5
2.2 Cache Memory	5
2.2.1 Cache Memory Organizations	6
3 Cache Simulator	9
3.1 Pixie	9
3.1.1 Operation of Pixie	10

3.1.2	Options	10
3.2	Trace Generator	11
3.2.1	Features	13
3.3	Dinero Cache Simulator	14
4	Performance Evaluation	17
4.1	Matrix Multiplication	17
4.2	SPEC92 Benchmarks	29
5	Conclusions	55
	List of References	56
	Bibliography	59

List of Figures

1	Different Types of Cache Organization	7
2	Block diagram of the Simulator	9
3	An example of a trace output file	12
4	Dinero cache Simulator output file	15
5	Miss percentage for matrix multiplication(128x128) vs blocksize for B1=8 & B2=16	18
6	Performance improvement for matrix multiplication vs datasize . . .	19
7	Performance improvement for matrix multiplication vs B2 for different datasizes and linesize(b1) = 8	21
8	Datamiss percentage for Matrix Multiplication vs B2 for different blocksizes	22
9	Datamiss percentage for Matrix Multiplication vs B1 for different blocksizes	23
10	Performance improvement for Matrix Multiplication vs B2 for different blocksizes	24
11	Performance improvement for Matrix Multiplication vs B1 for different blocksizes	25
12	Performance improvement for Matrix Multiplication vs B2 for different B1 (cache lines)	27
13	Performance improvement for Matrix Multiplication vs B1 for different B2	28
14	Datamiss percentage for GCC vs Cachesize (Assc. = 1)	31
15	Datamiss percentage for GCC vs Cachesize (Assc. = 2)	32

16	Performance improvement for GCC vs Cachesize	33
17	Datamiss percentage for COMPRESS vs Cachesize (Assc. = 1) . . .	35
18	Datamiss percentage for COMPRESS vs Cachesize (Assc. = 2) . . .	36
19	Performance improvement for COMPRESS vs Cachesize	37
20	Datamiss percentage for EQNTOTT vs Cachesize	38
21	Performance improvement for EQNTOTT vs Cachesize	39
22	Datamiss percentage for TOMCATV vs Blocksize	40
23	Performance improvement for TOMCATV vs Blocksize	41
24	Datamiss percentage for SWM256 vs Blocksize	43
25	Performance improvement for SWM256 vs Blocksize	44
26	Datamiss percentage for HYDRO2D vs Blocksize	46
27	Performance improvement for HYDRO2D vs Blocksize	47
28	Datamiss percentage for SU2COR vs Blocksize	49
29	Performance improvement for SU2COR vs Blocksize	50
30	Datamiss percentage for NASA7 vs Cachesize	52
31	Performance improvement for NASA7 vs Cachesize	53

Chapter 1

Introduction

1.1 Performance Analysis

Characterization of machines, by studying program usage of their architectural and organizational features, is an essential part of a design process. In order to evaluate the performance potential of any design, performance analysis on various architecture approaches have to be carried out. Evaluating the performance of cache-based computer systems is a difficult task because of the complexity of program behaviors. Locality property of an application and reuse factors have to be considered. Traditional performance evaluation can be broadly classified into three categories: Analytical Modeling, Simulation and Measurement.

1.2 Analytical Modeling

Analytical models provide a quick and insightful performance estimate of a given design[18]. By varying different input parameters over a wide range, analytical model is a good approach for a comparative study of the performance of different alternates of any design particularly cache design [19]. But the analysis and numerical results of analytical models are to some extent hypothetical and are not meant to predict performance of any realistic computer system.

In most cases, analytical models are used in the initial development of a new design whereas event-driven and trace-driven simulations are used to validate the design.

1.3 Measurement

Measurement is the only accurate and realistic performance evaluation. Since the cache design space is incredibly diverse with tens of independent design variables per level in a memory hierarchy, it is impossible to explore the entire design space in one study. Also the flexibility is limited which constrains the design methodology. To be able to do measurement, the system needs to be in existence. Therefore it is unsuitable for new designs

1.4 Simulation

1.4.1 Event-Driven Simulation

Event-Driven Simulation simulates activities of a system by generating random events according to a given distribution. Event-driven simulation can be carried out in various modes of which Time-Driven and Execution-driven Simulation are the most widely used [7]. Time-driven simulation is synchronous in the sense that all the system activities occur at discrete time intervals which are processor cycles.

1.4.2 Trace-Driven Simulation

In trace-driven simulation, one or more application programs are executed, usually interpretively, and a complete trace is collected from each. The trace typically contains all of the memory addresses referenced, as well as opcodes and possibly timing information. Traces can either be used directly, for example, to evaluate instruction set characteristics [21], or as input to an architectural simulator to predict the performance of different architectural variants. Such trace-driven simulation is most frequently used to study the behavior of cache memories [10].

The validity of trace-driven simulation relies on a crucial assumption: that perturbations to the trace data caused by the tracing process do not affect the simulation results. Unfortunately, it is nearly impossible to collect traces without perturbing program execution in some way. The most common perturbation is *execution dilation*. [24] validated the use of trace-driven simulation for multi-processors. Variability due to dilation and multiple runs appears to be small.

Trace-Driven simulation is based on actual traces of programs running on a system [4]. Therefore it provides the most reliable and accurate performance estimates for given programs on a given system. Trace-driven simulation may not capture the exact performance behavior of hypothetical architectures but, since, it reflect the actual hardware simulation of any design, it works as a good comparative study evaluation tool.

This thesis aims at evaluating the new prime-mapped cache design [12] through trace driven simulation . The analytical models and the event-driven simulation have been tested before and the new design is found to have a better performance over the conventional design. Traces are generated for a wide a range of numerical algorithms like matrix multiplication which forms an interesting case study for cache memories because locality is carried in three different loops by three different variables and SPEC benchmark programs [23] [13] which generate billions of references and which are a long running scientific applications benchmark set.

The traces are generated through a cache simulator which simulates direct-mapped/set-associative cache design and also the prime-mapped cache design. The traces are obtained from the object code of each algorithm which are fed to a trace counter called the Dinero-cache simulator. The parameters of the cache are changed to accommodate various configurations and the input data size is varied so as to reflect real cases.

1.5 Outline

The thesis is organized as follows:
Chapter 2 describes the various cache organizations including prime-map cache design. Chapter 3 describes the trace-driven cache simulator while chapter 4 discusses the plots and results. Finally chapter 5 concludes this thesis with a wrap-up and suggestions for future development.

Chapter 2

Architecture Study

2.1 Background

The fast computation of numerically intensive programs presents a challenge to memory system designers. Numerical program execution can be accelerated by pipelined arithmetic units [2], but to be effective, must be supported by high speed memory access. A cache memory is a well known hardware mechanism used to reduce the average memory access latency [6].

2.2 Cache Memory

Cache memories are high speed buffers which are inserted between the processors and memory to hold those portions of the contents of main memory which are currently in use [1]. They can be also inserted between main memory and mass storage. Since cache memories are typically as fast as CPU, they can reduce the effective memory access time if carefully designed and implemented [11]. This chapter discusses one of the characteristics of cache memories: mapping functions. Five mapping functions - direct, fully associative, set associative, sector mappings and prime-mapped are discussed.

2.2.1 Cache Memory Organizations

Cache is usually designed to be user-transparent. Therefore, in order to locate an item in the cache, it is necessary to have some function which maps the main memory address into a cache location. For uniformity of reference, both cache and main memory (MM) are divided into equal-sized units, called blocks in the memory and block frames in the cache. The placement policy determines the mapping function from the main memory address to the cache location.

There are basically five placement policies: *direct*, *fully associative*, *set associative*, *sector* and *prime* mappings.

- **Direct Mapping** In this scheme, block i of the memory maps into the block frame i modulo M of the cache, where M is the total number of cache blocks. When a physical memory address is generated for a memory reference, the block address field is used to address the corresponding block frame. The tag bit address is compared with the tag in the cache block frame. If there is a match, the information in the block frame is accessed by using the address field. Figure 1.a illustrates this organization.
- **Fully Associative** In this mapping, any block in memory can be in any block frame. When a request for a block is presented to the cache, all the map entries are compared simultaneously (associatively) with the request to determine if the request is present in the cache [17]. Although the fully associative cache eliminates the high block contention, it encounters longer access time because of the associativity of a large number of blocks. Figure 1.b illustrates the fully associative buffer.
- **Set Associative** This represents a compromise between direct and associative mapping organization. In this scheme, the cache is divided into S sets with M/S block frames per set, where M is the total number of block frames in the cache. A block i in memory can be in any block frame belonging to the set i modulo S , as shown in Figure 1.c

Figure 1.a Direct mapping cache organization

Figure 1.b Fully associative cache organization

Figure 1.c Set associative cache organization

Figure 1.d Sector mapping cache organization.

Figure 1: Different Types of Cache Organization

- **Sector Mapping** In this scheme, the memory is partitioned into a number of sectors, each composed of a number of blocks [15]. Similarly the cache is also divided into sector frames, each composed of a set of block frames. The memory requests are for blocks, and if a request is made for a block not in the cache, the sector to which this block belongs is brought into the buffer. The limitations are that that the mapping of blocks in a sector is congruent. Also, only the block that caused the fault is brought into the cache, and the remaining block frames in this sector frames are marked invalid thus wasting bandwidth. Figure 1.d illustrate the sector-cache organization.

- **Prime Mapping**

In a prime-mapped scheme, [12] each memory address, same as conventional cache-based computer system [5], is partitioned into three fields: $W = \log_2(\text{line size})$ bits of word address in a line(offset); $c = (\log_2(\text{number of sets} + 1))$ bits of index; and the remaining tag bits. The access logic of the prime-mapped cache consists of three components: data memory, tag memory and matching logic. Same as a set-associative cache, the data memory contains a set of address decoders and cached data; the tag memory stores tags corresponding to the cached lines; and the matching logic checks if the tag in an issued address matches the tag in the cache. The cache lookup process is exactly same as the set associative cache. However, the index field used to access the data memory is not just a subfield of the original address word issued by the processor since the modulus for cache mapping is not a 2's power any more [16]. it is the residue of the line address modulo a Mersenne number.

Chapter 3

Cache Simulator

This chapter describes the cache simulator used to simulate the two cache designs - set-associative-mapped and prime-mapped. The Simulator can be broadly divided into two parts: the first part called the XSIM[10] trace generator takes the pixiefied output of any executable code and generates traces for the second part, the DINERO[21] cache simulator to perform the actual simulation and report the results. The basic flow diagram is given in Figure 2.

The explanation for pixie, trace generator and the cache simulator is described below

3.1 Pixie

Generating traces for executable codes running into megabytes imposes severe constraints on the operating system. The trace counts for numerical algorithms

Figure 2: Block diagram of the Simulator

typically run into billions which cannot be stored in a hard copy. Hence, for trace-driven simulation to perform accurately, the input code is to be divided into several smaller sub-codes which can be accessed individually. The traces generated from these codes are of smaller size and hence can be recorded. The division of a bigger code into smaller codes is done by Pixie (a DECstation system utility).

3.1.1 Operation of Pixie

Pixie takes in an executable program from a DECstation compiler, partitions into basic blocks each of size 64k bytes and writes an equivalent program containing additional code that counts the execution of each basic block. A basic block is a region of the program that can be entered only at the beginning and exited only at the end. The input executable code is identified based on its magic number. Pixie exits on an undefined input magic number. The internal division of the code is done based on a dynamic stack allocation. Each block has a unique starting address with which it is identified. There can be correspondence within each block. To optimize the performance, pixie groups those instructions which can fit into the range of 16-bit displacement. An error will be generated if the offset exceeds 16 bits (signed). Pixie writes this output code into a default .pixie file extension. As the code is divided and information is to be stored as to their addresses, the pixified code is considerably larger than the input code. The branch instructions in the program determine the number of times each basic block in the program text is executed and the sequence in which the blocks are executed. Pixie also supports thirty two 32-bit general purpose registers which are used for data movements. All the operations are register-to-register except for load and store operations which are memory-to-register operations.

3.1.2 Options

In addition to generating address counts pixie also supports certain important features:

- Pixie defines a MIPS instruction set which is compatible with dec based risc machines.
- To allow for the individual blocks to be accessed, pixie maintains a file which gives the starting addresses for each of the blocks.
- Pixie supports the fortran 77 format statements by putting the original text into the translated output.
- To account for the trace references (used by the simulator), pixie enables the issue of memory references which enlarges the code considerably. Care must be taken in using this option as the branch offset may exceed 16 bits range on a subroutine call.
- In order to reduce the number of references generated pixie can issue only one memory reference for every N memory references, where N is an user defined number greater than 1.

Pixie doesn't work on programs that receive signals as the handler for address to the system calls is not translated. Also since the pixified code is considerably larger than the original code, conditional branches that used to fit in the 16-bit branch displacement field doesn't fit which generates a pixie error. This drawback is exposed by the perfectclub benchmark programs wherein the offsets are in the order of 18 bits.

3.2 Trace Generator

The output of the pixified code is fed to a trace generator. The XSIM trace generator reads each basic block of the pixified code and converts them into an assembly language code based on MIPS instruction set. This code is then assembled and converted into a trace output.

```
2 20d
2 211
0 1fc780
1 7ffcb0
2 213
2 217
0 1fc77c
1 7ffccac
```

Trace output file. This is an ASCII file with one LABEL and one ADDRESS per line. The rest of the line is ignored so that it can be used for comments.

LABEL =	0	read data
	1	write data
	2	instruction fetch
	3	unknown access type
	4	cache flush

0 <= ADDRESS <= ffffffff where the hexadecimal addresses are not preceded by "0x".

Figure 3: An example of a trace output file

Each trace output has two columns as shown in the Figure 3. The first column refers to whether its an instruction or data reference. For data references, read and write operations are distinguished. All other undefined operations are categorized under misc which includes flushing of cache. The Second column refers to the address accessed for instruction reference or the data for data reference. The starting address of the trace file can be specified by the user or set to a default address by the operating system. Since all other accesses are relative to this starting address it doesn't impose a burden on the user. The traces are generated for each individual blocks sequentially and are fed to the cache simulator as soon as they are generated. In order to optimize the performance, the traces are fed one at a time to the simulator rather than a whole block of traces as this may waste the processor time wherein the simulator is waiting for the trace generator to send a block of traces.

3.2.1 Features

- Xsim has provision to generate an entire trace file which can be subsequently fed to any cache simulator which takes in the same kind of input file. This facility is rarely used as the trace file may run into gigabytes. Instead as explained previously traces are fed one at a time without generating a trace file.
- Xsim has the ability to suppress tracing and just generate data files. This is equivalent to an assembler.
- To stop the trace generation after a fixed amount of time, Xsim has an option to stop the generator after N serial cycles are traced.
- To make it more user visible Xsim can generate traces starting from some fixed address.
- For comparative study, it may be needed to act only on a fixed amount of traces. This can be done by specifying both the starting and ending addresses

for trace generation thereby fixing the amount of traces generated and also avoiding accessing undefined addresses.

- To make it more user friendly, Xsim has a 9 level debugger which can trace the address accessed on. The debugging includes acting on new basic blocks, analyzing basic blocks, producing results for each of the basic blocks etc.

3.3 Dinero Cache Simulator

The traces generated by the xsim is fed into the Dinero cache simulator. The simulator takes as its inputs the organization of the cache like the unified cache size, instruction cache size, data cache size, block and sub-block sizes, associativity, write back policy etc. Once the cache parameters are fed into the simulator, it checks for discrepancies in the set-up like specifying a blocksize which is not equal to 2^c for some positive integer c or an undefined write back policy etc.

When the simulator recognizes that valid input parameters are specified, it initializes the address stack and starts fetching the traces. The address part is decoded and the tag and index fields are determined. The simulator looks for data access in the cache. On a miss (address tag not found in the cache), the main memory is accessed and the cache is updated. The data trace following the instruction trace is then acted upon. All the data and instruction references are recorded including read, write or misc accesses. The misses corresponding to each of the above cases are also filed. The address stack is continuously updated based on input specifications like the prefetching mode, flushing of cache etc. The results are written to an output file an example of which is given below.

Once the cache simulation is done, the output file is updated with the recorded values of instruction and data references. As can be seen from Figure. 4, the misses are calculated as a percentage of the total number of references for that category. Also the number of memory references is also given which when large degrades the cache performance. Since it is found that the simulator spends 35% to 50% of its

Ran Xsim on matrix64 - Mon Jan 4 14:12:19 1993

EXECUTION begins

CACHE (bytes) : blocksize = 32, sub-blocksize=0, Usize=0, Dsize=4096, Isize=4096.

POLICIES : assoc=1-way, replacement=1, fetch=d(1,0), write=c, allocate=w.

CTRL : debug=0, output=0, skipcount=0, maxcount=2147483000, Q=0.

Cache simulation begins

Total cycles = 18109095

Cache simulation completes.

Metrics (totals, fraction)	Access Type:					
	<u>TOTAL</u>	<u>Instrn</u>	<u>Data</u>	<u>Read</u>	<u>Write</u>	<u>Misc</u>
Demand Fetches	24267929 1.0000	5511950 0.2271	18755979 0.7729	13397127 0.5520	5358851 0.2209	1 0.0000
Demand Misses	1576511 0.0649	518674 0.0941	1057837 0.0564	915023 0.0683	142814 0.0266	1 1.0000

EXECUTION completes.

Figure 4: Dinero cache Simulator output file

time in scanning the input, a fast c function called "sscanxx" is used to interpret the input characters instead of the standard library function "sscanf". The function sscanxx runs about 6 times faster than sscanf.

Based on the access time of the cache and the memory (which can be set to values corresponding to actual hardware delays) and the penalty time for a miss which can be varied, the total execution time for any algorithm can be calculated. By varying the nature of the algorithms, and by studying the performance, one can achieve a reasonable knowledge of the cache design.

Two models of the simulator are developed. One for the Unprime-mapped cache and the other for the prime-mapped version. A comparative study of the cache performances over a wide range of varying parameters for distinct algorithms are carried out which are presented and analyzed in the next chapter.

Chapter 4

Performance Evaluation

4.1 Matrix Multiplication

Matrix Multiplication has been used to evaluate architecture design for a considerable longtime and hence has been included to analyze the prime-mapped design. Square matrices are used ranging from 64 X 64 to 256 X 256 long integers. Each matrix is divided into blocks(submatrices) of size $B1 \times B2$ and the algorithm was run on these blocks. An exhaustive study has been done by varying the blocksize, cachesize, datasize and the blocking factor and the results are plotted below.

Blocksize represents the number of bytes of data moved between cache and main memory in a single access. Figure.5 shows the datamiss percentages for varying blocksize. The cache size is kept constant at 4k bytes. B1 (blocking factor) is 8 and B2 is 16. In effect the matrices are multiplied as 8×16 matrices. The miss percentage decreases as the blocksize increases, since the probability of getting a match increases with more number of bytes to match. But the misses increase for blocksize of 32, as this represents a direct-mapped cache which generally performs poorer than a set-associative cache.

Figure.6 shows the improvement as a function of data size. The improvement is in absolute terms as the ratio of the misses for unprimed version over the primed version expressed as percentage. The performance improves as the datasize increase due to

Figure 5: Miss percentage for matrix multiplication(128x128) vs blocksize for B1=8 & B2=16

Figure 6: Performance improvement for matrix multiplication vs datasize

more number of line interferences. The prime version reduces the line interferences through the effective use of the cache space. The improvement decreases as the blocksize is increased due to less number of line interferences.

Figure.7 plots the variation in performance gain for B2. B2 represents the inner most loop of the algorithm and hence contributes mor eto the miss ratio than any other variable. The peak performance at B2 of 16 represents the pattern of the matrices suitable for this cache parameters. The performance degrades for B2 of 32 as the misses increase to more number of interferences. The plots for different datasizes indicate that the best performance is achieved for datasize 128x128 which is used to analyze the cache behavior in greater depth in Figure 8. which gives the miss percentages against B2 for different blocksizes. The increase in misses for increasing B2 is due to more number of block replacements for each multiplication of the matrix elements. The table for various linesizes show a decrease in misses due to reduction in line interferences for larger linesizes.

Figure 7: Performance improvement for matrix multiplication vs B2 for different datasizes and linesize(b1) = 8

Miss Percentages for Matrix Size 128x128						
Blocksize (bytes)	B2 (bytes) B1 = 8 bytes					
	8		16		32	
	unprime	prime	unprime	prime	unprime	prime
4	7.82	2.22	7.63	2.57	7.68	3.27
8	4.63	1.43	4.16	1.44	3.90	1.69
16	1.97	1.24	1.83	0.90	1.52	1.06

Figure 8: Datamiss percentage for Matrix Multiplication vs B2 for different blocksizes

B1 represents the number of times the inner most loop(once loaded) will be executed. Since B2 of 16 gives the best performance improvement for any B2, a variation in B1 over that value of B2 for miss percentages represent Figure 9. As for B2, the misses increase for increasing B1, (more replacements, more misses) and decrease for increasing linesizes due to less number of misses.

Miss Percentages for Matrix Size 128x128						
Blocksize (bytes)	B1 (bytes) B2 = 16 bytes					
	8		16		32	
	unprime	prime	unprime	prime	unprime	prime
4	7.63	2.57	7.47	2.45	7.30	3.04
8	4.16	1.44	4.57	2.89	4.43	3.62
16	1.83	0.90	2.92	2.09	2.93	2.56
32	4.58	2.95	5.34	3.96	5.29	3.56

Figure 9: Datamiss percentage for Matrix Multiplication vs B1 for different blocksizes

The performance gain when plotted against B2 is given in Figure 10. The performance degrades for increasing B2 due to more number of line interferences. Also increasing blocksizes reduce the gain due to less number of misses.

Figure 11 gives the corresponding increase in performance for B1. As for B2, the performance degrades for larger B1 (more replacements) and increases for smaller linesizes. (more chance of a miss).

Figure 10: Performance improvement for Matrix Multiplication vs B2 for different blocksizes

Figure 11: Performance improvement for Matrix Multiplication vs B1 for different block sizes

Figure 12 shows the effect of B1 on B2 as a factor for performance gain. decreasing B1 improves the performance as this reduces the number of times the blocks have to be replaced. A miss here will be propagated throughout the matrix multiplication. The peak gain for $B1 = 8$ and $B2 = 8$, is substantiated in Figure 13 which illustrates the effect of B2 on B1.

The discussion proves that the prime-version performs better than the un-primed version over various cache parameters, but the peak performance depends on the organization of the matrices as well. This brings out the intricacies in cache behavior that, no one particular architecture can ensure the best performance over the entire range of problem size.

Figure 12: Performance improvement for Matrix Multiplication vs B2 for different B1 (cache lines)

Figure 13: Performance improvement for Matrix Multiplication vs B1 for different B2

4.2 SPEC92 Benchmarks

SPEC (Standard Performance Evaluation Corporation) Benchmarks have become an important measure of CPU performance. Several factors, including strong industrial support for the System Performance Evaluation Consortium, the realistic nature of the benchmarks, and acceptable code portability, have led to the wide use of these programs for benchmarking purposes, and their consequent influence on system design.

SPEC Benchmarks are a selection of nontrivial programs chosen to standardize benchmarking and assembled to provide a standard set of realistic benchmarks for intersystem comparisons. The SPEC92 Benchmark suite consists of six integer-intensive C programs (compress, eqntott, espresso, gcc, sc and xliisp) and 14 floating-point intensive Fortran programs (alvinn, doduc, ear, fpppp, hydro2d, mdljdp2, mdljsp2, nasa7, ora, spice, su2cor, swm256, tomcatv and wave5). Only 3 C programs (compress, gcc and eqntott) and 5 Fortran programs (hydro2d, nasa7, su2cor, swm256 and tomcatv) are compiled and used for trace generation and architecture comparison. The reason for this is the low miss ratios for the other programs [23].

The SPEC Programs are compiled and run on DECstations that contained the Mips R3000 microprocessors running version 4.2a of the DEC Ultrix Operating System. Version 3.6.20 Fortran compiler is used for the floating point programs and version 2.0 of the C compiler. The cache size is varied from 4k to 32k bytes, block-size from 16 to 64 bytes for set sizes 1 and 2.

The SPEC benchmarks produce traces reaching several billions. When running the simulator for floating point fortran programs, it was observed during the trace generation that the simulator goes to an idle state when the benchmarks are run in full. After several experiments, it was found that by reducing the number of iterations for each of the benchmarks, which doesn't affect the individual algorithms but decrease the sample sizes, the trace generation and subsequently the simulation

can be carried out successfully. This reduced the amount of traces generated but since the objective is to compare the performance of the two cache designs on SPEC92, the modification is justified. The reason for the trace suspension needs to be sorted out yet.

Figure 15.1

QUC is a C/C++
 mixed sub-3 parallel
 16.3.13. plots of
 a well associated
 to availability of
 The plots for data
 large associated

Figure 15.2 shows
 plot of data remain
 effect the cache
 cache, the per
 to the time

Figure 14: Datamiss percentage for GCC vs Cachesize (Assc. = 1)

GCC is a GNU C compiler program which converts preprocessed files into optimized sun-3 assembly code which is written in C and cannot be vectorizable. Figure 14 & 15. plots the miss percentages for GCC for varying Cachesize and associativity 1 and associativity 2 respectively. The misses decrease for increasing cachesize due to availability of more data inside the cache thus increasing the probability of a hit. The plots for different associativities indicate that the miss percentage decrease for larger associativity as was proved for the matrix multiplication algorithm.

Figure 16. shows the performance improvement for the same parameters. The gain almost remains constant around 110 %. Variation in cachesize, hence, doesn't affect the cache performance in this case. When compared with 2-way set associative cache, the performance improvement of the prime-mapped cache varies with the cache size. Cache size change from 4 kbytes to 8 Kbytes reduces the number of line

Figure 15: Datamiss percentage for GCC vs Cachesize (Assc. = 2)

Figure 16: Performance improvement for GCC vs Cachesize

conflicts in the 2-way set-associative cache giving rise to the drop in performance improvement. However after cache size exceeds 8 Kbytes, the reduction in miss percentage is more for prime version than for the un-prime version thus showing an increase in performance gain.

Figure 1.7

Figure 17: Datamiss percentage for COMPRESS vs Cachesize (Assc. = 1)

COMPRESS is a C program which performs data compression on a 1 MB file using adaptive Lempel-Ziv coding. The variation in miss percentage as cachesize is varied for COMPRESS is shown in Figures 17 & 18 for different associativities. As for GCC, the misses decrease for increasing cachesize. The performance gain plotted in Figure 19. indicates that a cachesize of 8K bytes gives the lowest gain. For cachesizes, less than 8k bytes, the misses vary significantly (reducing for cachesize = 8k bytes more appreciably for un-primed than for primed version). For cachesizes = 8kbytes, the misses stabilize and hence produce more improvement. Also as the cachesize is increased, associativity plays a major role as can be seen from the curvature of the plots. The benchmark exhibits very high code locality and is not very sensitive to instruction cachesize. One 32KB direct-mapped cache had a miss ratio of less than one half of one percent. On the other hand, the benchmark is quite sensitive to data

Figure 18: Datamiss percentage for COMPRESS vs Cachesize (Assc. = 2)

Figure 19: Performance improvement for COMPRESS vs Cachesize

cache size; with miss ratios of eleven percent at 64KB and four percent at 256KB.

Figure 20: Datamiss percentage for EQNTOTT vs Cachesize

For EQNTOTT, a C benchmark program, which translates a boolean equation into a truthtable, the variation of associativity doesn't affect the cache performance significantly thus giving more or less the same miss percentages for associativities 1 & 2 as shown in Figure 20. EQNTOTT is an integer-intensive benchmark. The primary kind of computation performed is sorting. The plots follow the same pattern as the other SPEC benchmark results [23]. The straight line graphs for cachesizes in excess of 8Kbytes prove that the performance improvement achieved by increasing the cachesize is not worth the cost and the complexity involved in having a larger cache. The performance improvement given in Figure 21, indicate an increase in the rate of improvement for cachesizes upto 8kbytes.

Figure 21: Performance improvement for EQNTOTT vs Cachesize

Figure 22: Datamiss percentage for TOMCATV vs Blocksize

TOMCAT is a vectorized mesh generation program, written in Fortran using double precision floating point while SWM256 is a shallow water model that solves water equations using finite difference equations with a 256x256 grid. It is written in Fortran using Single precision floating point. The Tomcat input source code was modified to reduce the number of iterations from 100(as given in SPEC92) to 80. One iteration produces information about the elements of the 2 input fields which are in matrix form, there residuals which are obtained by working on these input elements and the error value which is the difference between the expected value and the result value. The number of iterations doesn't affect the program model except in the number of elements acted upon. The output file(obtained with 80 iterations) matches with the result file(from SPEC92), hence justifying the modification.

Figure 23: Performance improvement for TOMCATV vs Blocksize

The miss percentages for TOMCATV plotted against Blocksize are given in Figure 22. The miss percentages reduce as the blocksize is increased due to more number of data bytes available in the cache to match against. From the graph it can be seen there is very little decrease for blocksizes greater than 64 bytes for both the prime and the unprimed version. Hence the plot for performance improvement in Figure 23 shows a degradation for blocksizes greater than 64.

Figure 24: Datamiss percentage for SWM256 vs Blocksize

The input file to SWM256 object code is modified to reduce the number of iterations from 1200 to 400. These iterations are used to calculate the number of comparisons of the individual diagonal terms. The output file lists the total number of points in the X & Y directions, Grid spacing in X & Y directions, the time filter parameters and the cycle number which is the iteration value. Changing the iteration value effected the execution time of the model, but didn't affect the grid points which describe the model. The execution time changes due to the more number of values to be computed.

Figure 24 shows the miss percentage for SWM256 against Blocksize. The miss percentages reduce as the blocksize is increased due to more number of data bytes available in the cache to match against. From the graph it can be seen there is very little decrease for blocksizes greater than 128 bytes for both the prime and the

Figure 25: Performance improvement for SWM256 vs Blocksize

unprimed version. Hence the plot for performance improvement Figure 25. shows a degradation for blocksizes greater than 128. The input file to SWM256 object code is modified to reduce the number of iterations from 1200 to 400. These iterations are used to calculate the number of comparisions of the individual diagonal terms.

Figure 26: Datamiss percentage for HYDRO2D vs Blocksize

Hydro2d is an astrophysics application program which solves hydro-dynamical navier stroke equations to compute galacial jets. The input file is modified to change the number of timesteps from 400 to 100. This is the only number in the input file supplied with the benchmark, other input data are generated by the program itself. The output file specifies the time step, the GRID spacing, the viscosity factor and the execution time. The input file specifies the points to be computed. Changing this parameter affected the number of GRID points generated (from 400 to 100) and the viscosity factor. But the GRID points obtained upto 100 steps match with the result file from the benchmark and hence the program application is not changed.

Figure 26 gives the datamiss percentage as the blocksize is increased. Blocksizes greater than 128 bytes produce little improvement in the number of cache hits as the block is big enough to hold significant number of data bytes to produce very little

Figure 27: Performance improvement for HYDRO2D vs Blocksize

misses.

Figure 27 gives the corresponding performance improvement for blocksize variation. The performance shows a low at 64 bytes where the effective utilization of cache occurs and hence little improvement is achieved.

Figure 28: Datamiss percentage for SU2COR vs Blocksize

Su2cor computes masses of elementary particles. It is written in fortran using double precision floating point. The benchmark is scalable according to the number of configurations generated; this is the third number in the third record in the input data supplied with the benchmark. The number of iterations are reduced from 50 to 10. The output file specifies the frequency of correlation measurements, the matrix dimensions, the vector length and the current and the average value for each of the iterations. Reducing the iterations affected the overall average value of the Polyakov line averages, but the individual current measurements remained the same.

Figure 28 shows the miss percentage as the cachesize is increased. The miss percentages are high for smaller cachesize of 32 Kbytes but decrease as the cachesize is increased. Cachesizes of 128k and greater produce little improvement in data hits and represents poor utilization of cache space. The miss percentages for prime

Figure 29: Performance improvement for SU2COR vs Blocksize

version are about half of un-prime version.

The performance improvement vs cachesize (Figure 29) shows that rate of improvement decreases as the cachesize is increased. This is due to the decrease in misses as the cachesize is increased. The misses vary very little for larger cachesizes and hence the graph stabilizes for cachesize of 128 kbytes and greater.

Figure 30: Datamiss percentage for NASA7 vs Cachesize

Nasa7 is a collection of 7 kernals.

MXM - matrix multiply

CFFT2d - complex radix 2 FFT on 2D array

CHOLSKY - Cholesky decomposition in parallel on a set of input matrices

BTRIX - Block tridiagonal matrix solution along one dimension of a four dimensional array

GMTRY - Sets up arrays for a vortex method solution and performs gaussian elimination on the resulting arrays

EMIT - Creates new vortices according to certain boundary conditions

VPENTA - inverts 3 matrix pentadiagonals in a highly parallel fashion.

The input code is modified to reduce the number of variables from 6 to 5 and

Figure 31: Performance improvement for NASA7 vs Cachesize

array size from 7 to 5. The output file contains information about the time spent in each of the seven kernels and the floating point operations associated with that kernel. Reducing the variable size led to the optimization of different kernels and the array size reduced the level of complexity from 7(from SPEC92) to 5. This modification results in lower execution times for each of the seven internal kernels thereby reducing the overall traces generated.

Figure 30 plots the miss percentage as the cachesize is varied. The cachesize is varied from 64 Kbytes to 256 Kbytes as the traces generated are very high around 9 billions. The miss percentage reduce as the cachesize is increased due to less number of line conflicts for larger cachesizes. Figure 31 shows the performance improvement for the same cache size variation. The performance shows a increasing gain as the cachesize is made large. This is due to the less number of line conflicts in prime-version compared to unprimed version.

The SPEC results prove that the prime-version, when giving better performance gains, tend to stabilize over the range of 8k byte cachesize and varying associativities when reducing the miss percentages, have little impact on the cache performance.

Chapter 5

Conclusions

In this thesis, a new cache organization, Prime-mapped Cache Design was evaluated. An existing conventional Cache Simulator is modified to reflect the new design. The design is evaluated using trace generation through the XSIM trace generator, pixie and the Dinero cache simulator. Numerous programs and algorithms are compiled and used to generate the traces. Programs include Matrix Multiplication and the SPEC benchmarks. The SPEC benchmarks provide a valuable source as they are used to evaluate a similar design. Simulation is done for different cache sizes by varying the blocksize and the associativity.

Results are obtained for both the unprimed and the primed version. A comparison shows that the prime version shows a performance improvement ranging from 50% to 150% in the miss ratios depending on the algorithm used to evaluate it. Even though prime version cache shows considerable improvement, the large variation in the performance reflects the complexity and unpredictable nature of the cache design. An architecture which gives good performance for one algorithm may perform poorly when used for a different algorithm. The wide range of programs used to evaluate the new design takes this into consideration and the results show that the prime-mapped cache design always gives a better performance than unprime-mapped design for scalar processors.

References

- [1] J.L. Hennessy and D.A. Patterson, *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 1990.
- [2] M.S. Lam, E.E. Rothberg and M.E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proc. of Arch. Supp. for Prog. Lang. and Opr. Sys.*, pp. 63-74, April 1991.
- [3] J.W.C. Fu, J.H. Patel and B.L. Janssens, "Stride Directed Prefetching in Scalar Processors," in *Proc. 19th. Int'l Symp. on Comp. Arch.*, pp. 102-110, 1992.
- [4] A. Borg, R.E. Kessler and D.W. Wall, "Generation and Analysis of very long address traces," *Proc. 17th. Int'l Symp. on Comp. Arch.*, pp. 270-279, May 1990.
- [5] M.D. Hill, "A case for direct_mapped caches," *IEEE Computer*, pp. 25-40, Dec. 1988.
- [6] A.J. Smith, "Cache Memories," *ACM Comp. Surveys*, vol 18, no. 3, pp. 473-530, Sept. 1982.
- [7] C.B. Stunkel and W.K. Fuchs, "TRAPEDS: Producing traces for multicomputers via execution driven simulation," *Proc. ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp. 70-78, May 1989.
- [8] H.S. Stone, *High Performance Computer Architecture*. Addison-Wesley, 1990.
- [9] K. So and V. Zecca, "Cache performance of vector processors," in *Proc. 15th. Int'l Symp. on Comp. Arch.*, pp. 261-268, 1988.

- [10] M.D. Hill and D.N. Pnevmatikatos, "Cache Performance of the Integer SPEC Benchmarks on a RISC," Computer Architecture News, Vol. 18, No.2, June 1990, pp. 53-68.
- [11] D. Gannon, W. Jalby and K. Gallivan, "Strategies for cache and local memory management by global program transformation," in Int'l conf. on Supercomputing, 1987.
- [12] Q. Yang and L.W. Yang, "A novel cache design for vector processing," the 19th Int'l Symp. on Computer Architecture, May 1992. Gold Coast, Australia.
- [13] M. Berry, et. al., "The Perfect Cub Benchmarks: Effective Performance Evaluation of Supercomputers," Int'l Journal for Supercomputer Applications, Fall 1989.
- [14] J.W.C. Fu and J.H. Patel, "Data prefetching in multiprocessor vector cache memories," in Proc., 18th Int'l Symp. on Comp. Arch., pp. 54-63, 1991.
- [15] P. Budnik and D.J. Kuck, "Organization and use of parallel memories," IEEE Trans. on Computers, pp. 1566-1569, Dec. 1971.
- [16] D.H. Lawrie and C.R. Vora, "The prime memory system for array access," IEEE Trans. on Computers, vol. C-31, pp. 435-441, May 1982.
- [17] K. Hwang and F.A. Briggs, Computer Architecture and Parallel Processing. McGraw-Hill, 1989.
- [18] Q. Yang, L. Bhuyan and B.C. Liu, "Analysis and comparison of cache coherence protocols for a packet switched multiprocessor," IEEE Trans. on Computers, vol. 38, pp. 1143-1153, August 1989.
- [19] M. Dubois and F. Briggs, "Effect of cache coherency in multiprocessors," IEEE Trans. on Comput., vol. C-31, pp. 1083-1099, Nov. 1982.

- [20] J. Archibald and J.L Baer, "Cache coherence protocols: Evaluation using multiprocessor simulation model," *ACM Trans. on Comput. Systems*, vol. 4, pp. 273-298, November 1986.
- [21] Mark D. Hill and Alan J. Smith, "Evaluating Associativity in CPU Cache Memories," *IEEE Trans. on Computers*, Vol. 38, No. 12, Dec. 1989, pp. 1612-1630.
- [22] Alan J. Smith, "Line(Block) Size Choice for CPU Cache Memories," *IEEE Trans. on Computers*, Vol. C-36, No. 9, Sept. 1987, pp.1063-1075.
- [23] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos and A.J. Smith, "Cache Performance of the SPEC92 Benchmark Suite," *IEEE Micro*, Vol. 13, No. 4, Aug. 1993, pp. 17-27.
- [24] Eric J. Kolding, Susan J. Eggers and Henry M. Levy, "On the Validity of Trace-Driven Simulation for Multiprocessors," *ACM Trans. on Comput. Systems*, vol. 6, July 1991, pp 244-250.

Bibliography

- Archibald, J. and Baer, J.L., "Cache coherence protocols: Evaluation using multiprocessor simulation model," *ACM Transactions on Computer Systems*, vol. 4, pp. 273-298, November 1986.
- Borg, A., Kessler, R.E. and Wall, D.W., Generation and Analysis of very long address traces, Proceedings 17th. International Symposium on Computer Architecture, pp. 270-279, May 1990.
- Budnik P. and Kuck, D.J., "Organization and use of parallel memories," *IEEE Transactions on Computers*, pp. 1566-1569, Dec. 1971.
- Dubois, M. and Briggs, F., "Effect of cache coherency in multiprocessors," *IEEE Transactions on Computers*, vol. C-31, pp. 1083-1099, Nov. 1982.
- Gannon, D., Jalby, W. and Gallivan, K., Strategies for cache and local memory management by global program transformation, International Conference on Supercomputing, 1987.
- Gee, J.D., Hill, M.D., Pnevmatikatos, D.N. and Smith, A.J., "Cache Performance of the SPEC92 Benchmark Suite," *IEEE Micro*, Vol. 13, No. 4, Aug. 1993, pp. 17-27.
- Hennessy, J.L. and Patterson, D.A., Computer Architecture, A Quantitative Approach, Morgan Kaufmann, 1990.
- Hill, M.D., "A case for direct mapped caches," *IEEE Computer*, pp. 25-40, Dec. 1988.

- Hill, M.D. and Pnevmatikatos., "Cache Performance of the Integer SPEC Benchmarks on a RISC," Computer Architecture News, Vol. 18, No. 2, june 1990, pp. 53-68.
- Hill, M.D. and Smith, A.J., "Evaluating Associativity in CPU cache Memories," IEEE Transactions on Computers, Vol. 38, No. 12, Dec. 1989, pp. 1612-1630.
- Hwang, K. and Briggs, F.A., Computer Architecture and Parallel Processing, McGraw-Hill, 1989.
- Koldinger, J. Eric., Eggers, J. Susan and Levy M. Henry., "On the Validity of Trace-Driven Simulation for Multiprocessors," ACM Transactions on Computer Systems, vol. 6, July 1991, pp 244-250.
- Lam, M.S., Rothberg, E.E. and Wolf, M.E., "The cache performance and optimizations of blocked algorithms," Proceedings of Architecture Support for Programming Languages and Operating Systems, pp. 63-74, April 1991.
- Lawrie, D.H. and Vora, C.R., "The prime memory system for array access," IEEE Transactions on Computers, vol. C-31, pp. 435-441, May1982.
- Patel, J.H. and Fu, J.W.C., "Data prefetching in multiprocessor vector cache memories," Proceedings 18th International Symposium on Computer Architecture, pp. 54-63, 1991.
- Smith, A.J., "Cache Memories," ACM Computer Surveys, vol 18, no. 3, pp. 473-530, Sept. 1982.
- Smith, A.J., "Line(Block) Size Choice for CPU Cache Memories," IEEE Transactions on Computers, Vol. C-36, No. 9, Sept. 1987, pp.1063-1075.
- So, K and Zecca, V., "Cache performance of vector processors," Proceedings 15th. International Symposium on Computer Architecture, pp. 261-268, 1988.
- Stone, H.S., High Performance Computer Architecture, Addison-Wesley, 1990.

- Stunkel, C.B. and Fuchs, W.K., "TRAPEDS: Producing traces for multicomputers via execution driven simulation," Proceedings ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, pp. 70-78, May 1989.
- Yang, Q., Bhuyan, L. and Liu, B.C., "Analysis and comparison of cache coherence protocols for a packet switched multiprocessor," IEEE Transactions on Computers, vol. 38, pp. 1143-1153, August 1989.
- Yang, Q and Yang, L.W., " A novel cache design for vector processing," 19th International Symposium on Computer Architecture, MAY 1992. Gold Coast, Australia.