# Improving Cache performance using SSMT

Vaibhav Agrawal
vaibhav@ece.utexas.edu

Viswanath Sankaranarayanan
viswa@cerc.utexas.edu

$12^{th}$ May, 1999

**Abstract**

*The average clock speed of processors have improved at a vastly greater rate than the memory access time. This will lead to a scenario where a cache miss will cost thousands of cycles in the future. Minimizing cache misses will therefore be crucial in the performance of computer systems. In this project, we probe the possibility of using a Simultaneous Subordinate Multithread (SSMT) to run a highly effective cache replacement policy during a miss and therefore, to minimize future misses that may occur. We also propose the concept of a pointer cache, which can be filled up by an SSMT, and which may help reduce the misses that occur due to the Object Oriented development environment of today. We first give an introduction to SSMT. We then study to see if there is anything to be gained by running such an SSMT based replacement policy. We finally propose the idea of the pointer cache (P-Cache), and give the various optimizations that can be carried to improve its performance.*

# 1 Introduction

In this section, we describe Simultaneous Subordinate Microthreading (SSMT) and how this can be applied to cache replacement policy to improve performance.

## 1.1 SSMT

The goal of SSMT[1] unlike Multithreading is to use additional threads to improve the performance of a single primary thread. These additional threads are supplied in the form of microcode, and are referred to as *microthreads*. Figure 1[1] shows an SSMT machine. Microthread routines are code sequences written in an internal instruction format of the processor. They can be stored on-chip in a microcode RAM (the *microRAM*). When certain processor events occur, a microthread is *spawned*, and corresponding microthread instructions are executed. Microthread optimizations have the advantage of encoding sophisticated algorithms without any concern for hardware complexity.

[1]Courtesy[1]

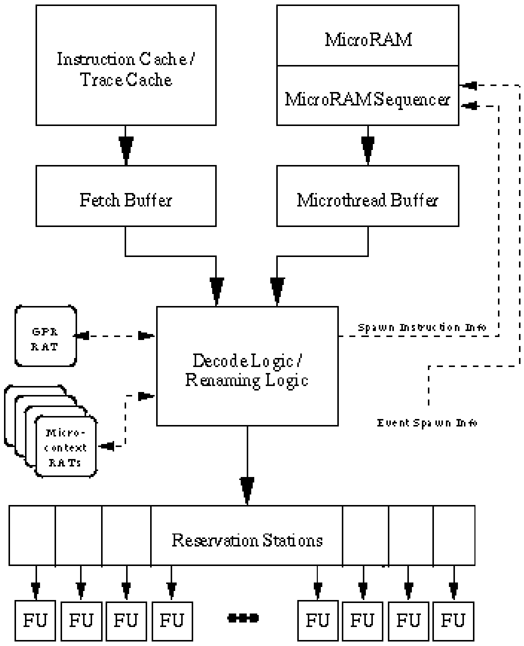


Figure 1: Block diagram of an SSMT machine.

### 1.2 Cache replacement problem

Processor clock speeds have improved vastly over the years, but memory latency has not kept up with it. Thus, in future, a cache miss will cost thousands of cycles. To amortize this, the number of cache misses will have to be reduced. This comes down to the point where we would have to look at some cache replacement policy other than the conventional. Sophisticated cache replacement policies have been proposed, but the bane of them all have been the fact that they cannot be easily implemented in hardware. By far the most popular replacement policy is LRU, but it is optimal only when loop lengths are approximately equal to the LRU stride length[2]. This restriction on the replacement scheme has resulted in miss rates that cannot be lowered any further. When the miss penalty reaches thousands of cycles, a situation may occur wherein the total execution time of programs may be dominated by the miss penalty. This project is an attempt to look into this problem and to come up with some mechanism that would lessen the miss rate and thereby reducing the total miss penalty.

## 2 Cache Replacement - SSMT

In this section we explain the idea[4] of using SSMT for cache replacement policy. Then we explain the simulations that we ran to validate the claim and finally, we present the results of the simulations.

## 2.1 The Idea

When a cache miss occurs we typically have an idle time of thousands of cycles. In this period we can spawn a micorothread that runs and decides which of the cache blocks have to be replaced by the incoming block. The idea is that a memory access request has to go out immediately, but a lot of time can be wasted thinking about which block should be replaced.

## 2.2 Is there any gain?

Before starting to implement a sophisticated cache replacement policy as a microthread, we first see if there is enough gain to be had. The best that we can do today is LRU. The best that we can ever do is OPT. OPT is when we replace the block that will be used farthest in future. Obviously it is impossible to implement OPT, except in simulations. The comparison between LRU and OPT gives us a indication of how much maximum gain we can get, and answer the question, *"is it worthwhile to look at a new replacement policy?"*

## 2.3 Feasibility Study

To determine the maximum gain that one could get by such a scheme, wrote a simulator to do both LRU and OPT replacement, and then we ran 3 of the SPEC95 benchmarks for a variety of cache sizes and configurations. Implementing and validating the LRU replacement was not too difficult. We wrote the simulator in C++, and used STL for the data structures. We generated the traces of the SPEC benchmarks using the Shade Analyzer[5] from Sun microsystems. The Shade suite also has a general purpose cache simulator and we validated our cache simulator against it, and the results were identical. We then implemented the OPT replacement policy. This basically entailed running through the traces twice and remembering the next occurrence of each block. We found that this was very inefficient, and so we modified this into a single pass simulator. We did the second pass (the pass where we determine the next occurrence) over the data in memory. To validate the OPT policy we took small traces and verified that the simulator was behaving as expected. Also, an indirect way of verifying this was to see if OPT replacement always gave better results than LRU.

## 2.4 Results

We ran the test for a block size of 16, 32 and 64 bytes. Further we ran it for a 2, 4 and 8 way associative cache from 1K to 512K as the cache size. As the size of the trace of the SPEC benchmark was large, we limited all the traces to a maximum of 7 million references. In the plots we show the results for a 32 byte block size and 3 SPEC benchmarks *compress*, *go* and *fpppp*. The complete results are attached in the appendix. We find from our simulations that OPT does perform better than LRU. This improved performance is more pronounced when the cache size is small. The *compress* instruction trace, has very few unique references and hence has anamolous behaviour. As the size of the cache increase, the gain tends to lessen.

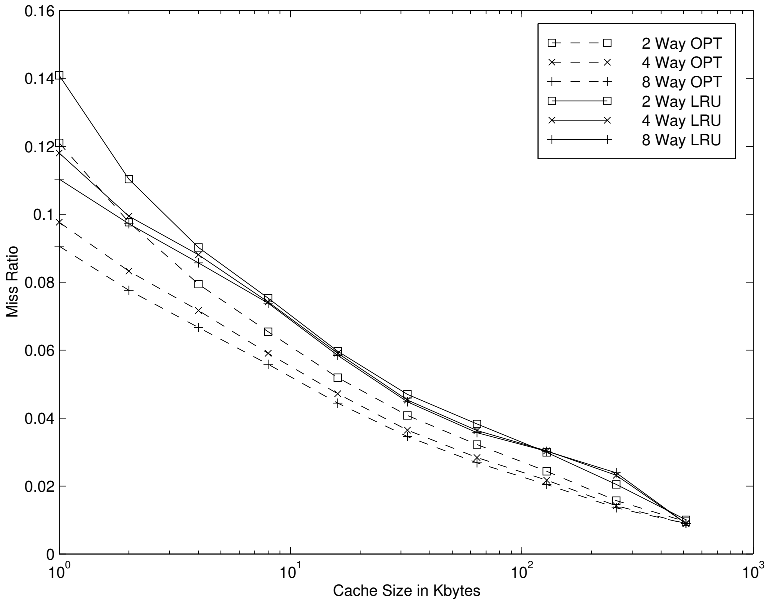


Figure 2: Data cache miss Ratio for *compress*

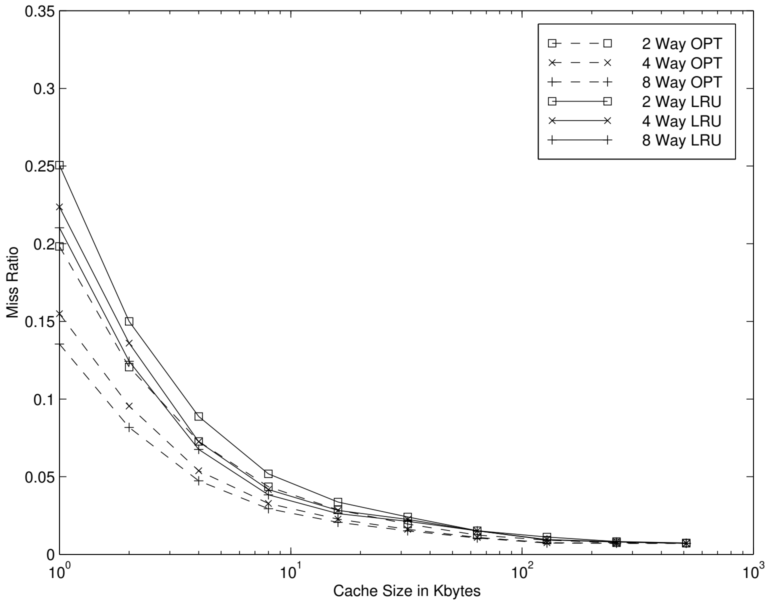


Figure 3: Data cache miss Ratio for *go*

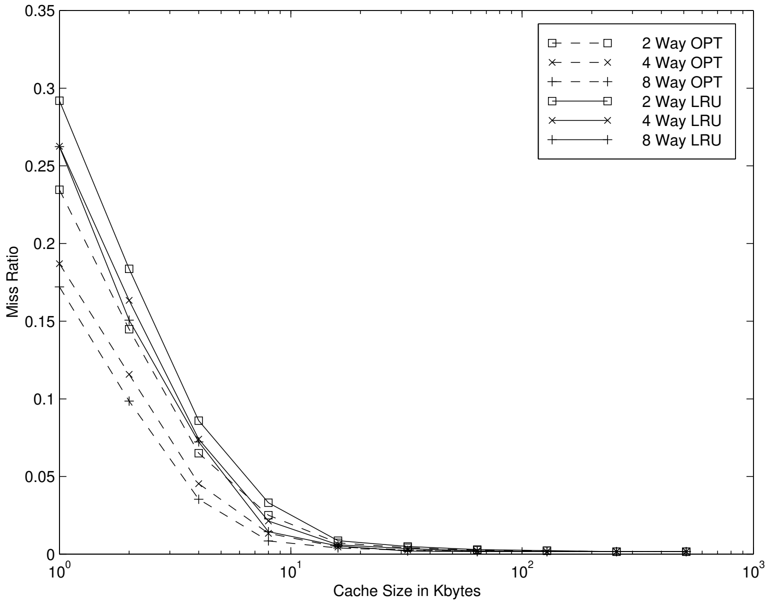


Figure 4: Data cache miss Ratio for *fpppp*

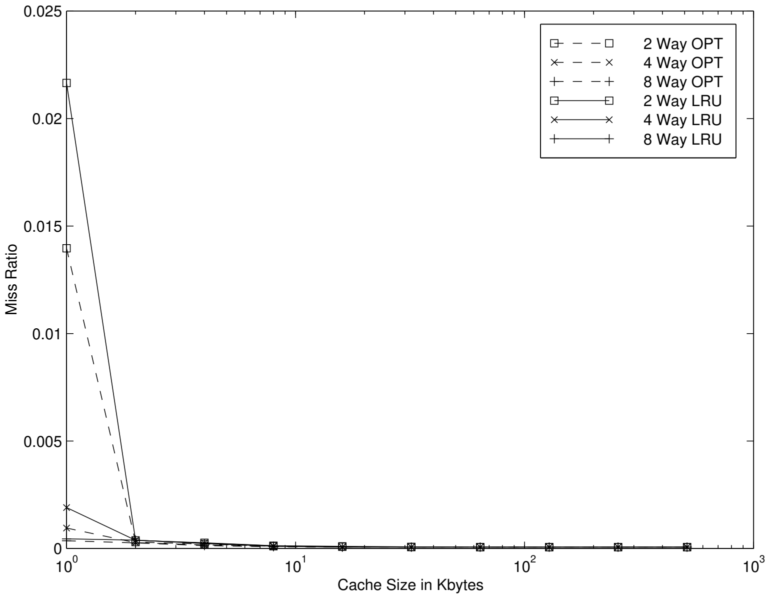


Figure 5: Instruction cache miss Ratio for *compress*

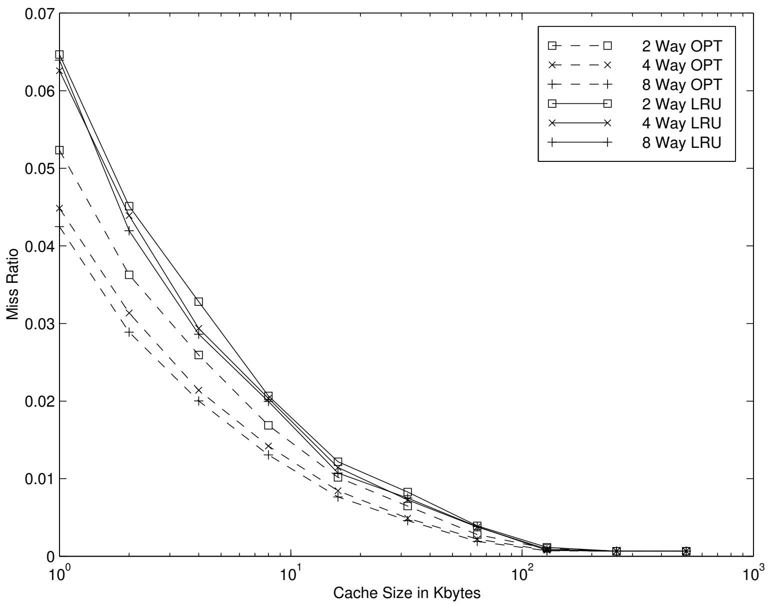


Figure 6: Instruction cache miss Ratio for *go*

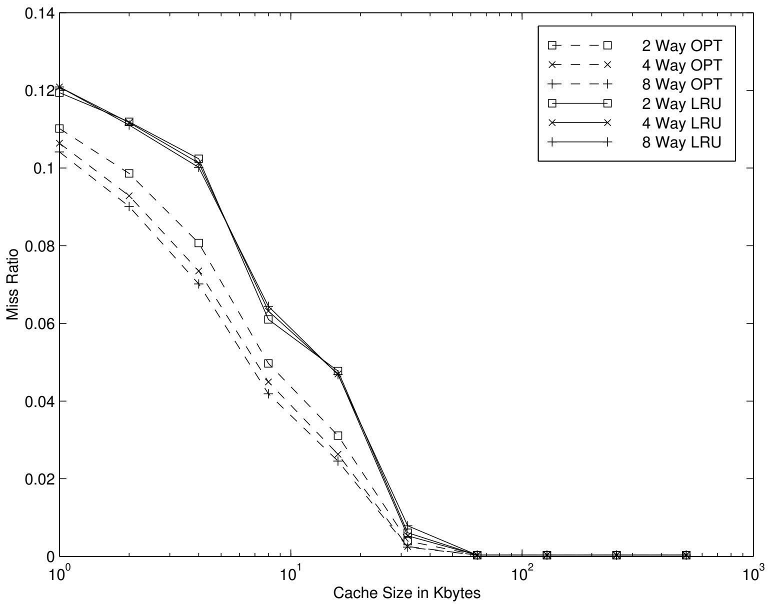


Figure 7: Instruction cache miss Ratio for *fpppp*

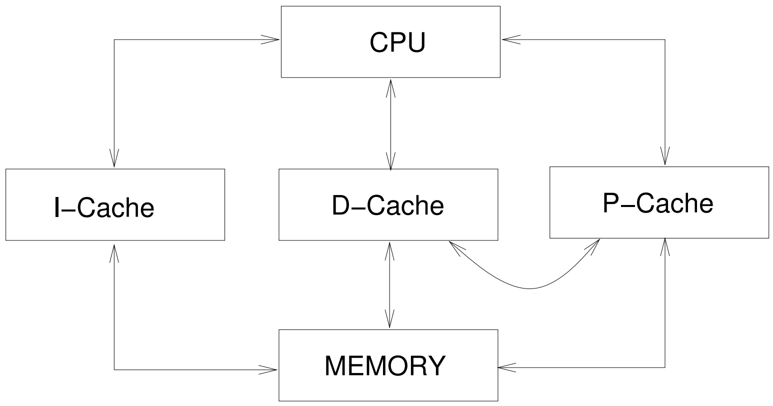


Figure 8: Memory hierarchy with P-Cache.

# 3 The P-Cache

In this section we propose a new cache idea called the Pointer cache (P-Cache). We first give a motivation for this and then describe its architecture. Finally, we also give a list of optimizations that can be done to improve its performance.

## 3.1 Why a P-Cache?

Today most of the software development occurs in the object-oriented programming languages, and typically we see that these have a lot of pointer oriented operations like *virtual table lookup* and *garbage collection* by reference counting. Typically, data caches handle pointers and the data they point to as separate entities. This could lead to a lot of cache misses. We propose a mechanism by which the pointer and its data will be treated as connected entities so that when a pointer is thrown out of the cache, its associated data will also be thrown out.

## 3.2 P-Cache Architecture

Figure 8 gives the block diagram of a memory with P-Cache. In addition to the data cache and an instruction cache, we propose to incorporate a pointer cache also in the memory hierarchy. Whenever a cache miss occurs either in the data or in the instruction cache, we run a microthread that dereferences all the pointer entries in the data cache and loads the associated data into the pointer cache (P-Cache). Whenever a pointer is dereference in the execution of a program, the value is first probed in the P-Cache, before going to memory. The assumptions made here are as follows:

- There exists some mechanism of determining whether a value in the data cache is a pointer to another data.
- There exists some mechanism of sending request after request to the memory before the first one has been serviced.

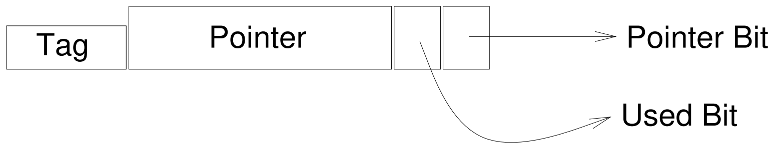


Figure 9: Modifications to data cache block.

| Pointer Bit | Used Bit | Meaning |
|---|---|---|
| 0 | 0 | Entry is not a pointer |
| 1 | 0 | Entry is a pointer and the value is to be loaded into P-cache |
| 0 | 1 | Entry is a pointer and the value has been loaded into P-cache |
| 1 | 1 | Illegal state |

Table 1: Status bits of the D-Cache

We find that both these assumptions are reasonable, as there needs to be only a small change in the ISA to solve the first problem, and with the advent of rambus, the second problem is also ameliorated.

### 3.3 Optimizations

Figure 9 shows the modification to the data cache block to incorporate a P-Cache. The *pointer bit* is used to signify, if the entry in the data cache is a pointer value or not. The *used bit* is used to signify that the data pointed to has already been loaded into the P-Cache. Table 1 gives the meaning of the status bits in a data cache entry. The optimizations that can be made to the proposed architecture are

- Typically a data cache is 64K in size and we have only 1000 cycles to run the microthread. To minimize the effort to go through all the entries in the data cache, we logically OR the pointer bits of all the blocks in a set and have a pointer bit for the set. We go through the entries in the block only if the pointer bit for the set is *set.* This would reduce the overall execution time of the microthread.

- The used bit can be used to identify a pointer and therefore, once a data is loaded into the P-cache, its pointer's *used bit* can be set to 1 and the *pointer bit* to 0. This again would reduce the microthread running time.

- Finally, whenever a pointer entry from the data cache is thrown out, if its value was loaded into the P-cache, then this entry is also thrown out. This crucial improvement gives the notion of a connection between the pointer and the data it points to.

## 4 Conclusion

The results of the simulations we ran indicate that there is a sizeable gain that can be had by running a microthread and by optimizing the cache replacement policy. As expected we find that OPT replacement policy performs better that LRU. But, as the cache size increases, the law of diminishing returns seems to take over and hence both of them perform about the same. The application of cache replacement microthread, is therefore more applicable the sizes of the conventional L2 cache. Various cache replacement policies[2] have been proposed, and these can be coded as a microthread to yield better performance than LRU. Finally, we have proposed the concept of a pointer cache and how a microthread can be used to fill it up and reduce the misses due to pointer operations.

## 5 Future Work

The sophisticated cache replacement policies would have to be implemented in the simulator and benchmarks would have to be run to asess the gain that we can get by these microthreads in practice. The P-Cache would have to be implemented in software and simulations run on them with object oriented benchmarks to asess the amount of gain that we would get by running such a microthread.

## References


[1] R.S. Chappel, J. Stark, S.P. Kim, S.K. Reinhardt, Y.N. Patt, "Simultaneous Subordinate Microthreading (SSMT)," *Proc. 26th International Symposium on Computer Architecture*, Altanta, May, 1999.

[2] Y. Smaragdakis, S. Kaplan, P. Wilson, "EELRU: Simple and Effective Adaptive Page Replacement," *ACM SIGMETRICS Conference*, May 1999.

[3] D.A. Patterson and J.L. Hennessy, *Computer Architecture: A Quantitative Approach*, ISBN 1-55860-329-8, 1996.

[4] Y.N. Patt, *Talk on Computer Architecture*, Dept. of Electrical and Computer Engg., University of Texas, Austin, February 1999.

[5] Sun Microsystems Inc. *Introduction to Shade*, Product documentation, June 1997.

[6] E.S. Tam, J.A. Rivers, V. Srinivasan, G.S. Tyson, E.S. Davidson, "Evaluating the performance of Active Cache Management Schemes," *Proc. ICCD-98*, pp. 368-375.

[7] T.M. Chilimbi, B. Davidson, J.R. Larus, "Cache-Consious Structure Definition," *Proc. of the ACM SIGPLAN '99.*

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.2319 | 0.2145 | 0.1194 | 0.1102 | 0.0628 | 0.0578 |
| 2K | 0.2184 | 0.1926 | 0.1119 | 0.0986 | 0.0588 | 0.0516 |
| 4K | 0.2006 | 0.1574 | 0.1024 | 0.0807 | 0.0535 | 0.0042 |
| 8K | 0.1185 | 0.0966 | 0.0610 | 0.0496 | 0.0320 | 0.0259 |
| 16K | 0.0946 | 0.0614 | 0.0477 | 0.0311 | 0.0242 | 0.0158 |
| 32K | 0.0116 | 0.0075 | 0.0061 | 0.0039 | 0.0033 | 0.0021 |
| 64K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 128K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 256K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 512K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |

Table 2: Miss ratio for *fpppp*, 2-way associative instruction cache

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.2.53 | 0.1074 | 0.1209 | 0.1064 | 0.0637 | 0.0560 |
| 2K | 0.2183 | 0.1815 | 0.1117 | 0.0929 | 0.0584 | 0.0486 |
| 4K | 0.1987 | 0.1438 | 0.1013 | 0.0735 | 0.0529 | 0.0384 |
| 8K | 0.1225 | 0.0876 | 0.0632 | 0.0449 | 0.0337 | 0.0236 |
| 16K | 0.0934 | 0.0521 | 0.0470 | 0.0263 | 0.0237 | 0.0133 |
| 32K | 0.0100 | 0.0048 | 0.0053 | 0.0026 | 0.0030 | 0.0014 |
| 64K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 128K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 256K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 512K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |

Table 3: Miss ratio for *fpppp*, 4-way associative instruction cache

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.2353 | 0.2030 | 0.1209 | 0.1042 | 0.0639 | 0.0472 |
| 2K | 0.2179 | 0.1762 | 0.1111 | 0.0901 | 0.0582 | 0.0365 |
| 4K | 0.1964 | 0.1374 | 0.1002 | 0.0702 | 0.0528 | 0.0219 |
| 8K | 0.1251 | 0.0822 | 0.0644 | 0.0419 | 0.0349 | 0.0125 |
| 16K | 0.0931 | 0.0486 | 0.0468 | 0.0246 | 0.0236 | 0.0013 |
| 32K | 0.0148 | 0.0044 | 0.0079 | 0.0024 | 0.0046 | 0.0002 |
| 64K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 128K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 256K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |
| 512K | 0.0006 | 0.0006 | 0.0003 | 0.0003 | 0.0002 | 0.0002 |

Table 4: Miss ratio for *fpppp*, 8-way associative instruction cache

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.1051 | 0.0855 | 0.0647 | 0.0523 | 0.0439 | 0.0345 |
| 2K | 0.0759 | 0.0613 | 0.0451 | 0.0363 | 0.0289 | 0.0023 |
| 4K | 0.0557 | 0.0443 | 0.0328 | 0.0259 | 0.0201 | 0.0160 |
| 8K | 0.0360 | 0.0294 | 0.0207 | 0.0169 | 0.0125 | 0.0102 |
| 16K | 0.0215 | 0.0178 | 0.0122 | 0.0102 | 0.0072 | 0.0061 |
| 32K | 0.0145 | 0.0112 | 0.0083 | 0.0065 | 0.0050 | 0.0039 |
| 64K | 0.0068 | 0.0049 | 0.0039 | 0.0028 | 0.0024 | 0.0017 |
| 128K | 0.0021 | 0.0018 | 0.0012 | 0.0010 | 0.0007 | 0.0006 |
| 256K | 0.0013 | 0.0013 | 0.0007 | 0.0007 | 0.0004 | 0.0004 |
| 512K | 0.0013 | 0.0013 | 0.0007 | 0.0007 | 0.0004 | 0.0004 |

Table 5: Miss ratio for *go*, 2-way associative instruction cache

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.1012 | 0.0733 | 0.0626 | 0.0449 | 0.0417 | 0.0300 |
| 2K | 0.0735 | 0.0532 | 0.0439 | 0.0314 | 0.0282 | 0.0198 |
| 4K | 0.0506 | 0.0369 | 0.0294 | 0.0214 | 0.0181 | 0.0132 |
| 8K | 0.0351 | 0.0247 | 0.0204 | 0.0142 | 0.0125 | 0.0087 |
| 16K | 0.0201 | 0.0148 | 0.0114 | 0.0084 | 0.0068 | 0.0051 |
| 32K | 0.0130 | 0.0086 | 0.0073 | 0.0049 | 0.0042 | 0.0029 |
| 64K | 0.0064 | 0.0038 | 0.0038 | 0.0022 | 0.0024 | 0.0014 |
| 128K | 0.0016 | 0.0014 | 0.0009 | 0.0008 | 0.0006 | 0.0014 |
| 256K | 0.0013 | 0.0013 | 0.0007 | 0.0007 | 0.0004 | 0.0004 |
| 512K | 0.0013 | 0.0013 | 0.0007 | 0.0007 | 0.0004 | 0.0004 |

Table 6: Miss ratio for *go*, 4-way associative instruction cache

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.1019 | 0.0691 | 0.0639 | 0.0425 | 0.0429 | 0.0285 |
| 2K | 0.0695 | 0.0491 | 0.0420 | 0.0289 | 0.0273 | 0.0182 |
| 4K | 0.0497 | 0.0346 | 0.0286 | 0.0200 | 0.0176 | 0.0123 |
| 8K | 0.0346 | 0.0228 | 0.0200 | 0.0131 | 0.0122 | 0.0079 |
| 16K | 0.0190 | 0.0135 | 0.0107 | 0.0077 | 0.0644 | 0.0046 |
| 32K | 0.0136 | 0.0081 | 0.0075 | 0.0046 | 0.0043 | 0.0028 |
| 64K | 0.0063 | 0.0032 | 0.0038 | 0.0019 | 0.0025 | 0.0012 |
| 128K | 0.0014 | 0.0013 | 0.0008 | 0.0007 | 0.0005 | 0.0004 |
| 256K | 0.0013 | 0.0013 | 0.0007 | 0.0007 | 0.0004 | 0.0004 |
| 512K | 0.0013 | 0.0013 | 0.0007 | 0.0007 | 0.0004 | 0.0004 |

Table 7: Miss ratio for *go*, 8-way associative instruction cache

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.0142 | 0.0091 | 0.0217 | 0.0140 | 0.0236 | 0.0169 |
| 2K | 0.0006 | 0.0005 | 0.0004 | 0.0003 | 0.0002 | 0.0002 |
| 4K | 0.0004 | 0.0003 | 0.0003 | 0.0002 | 0.0002 | 0.0002 |
| 8K | 0.0002 | 0.0002 | 0.0001 | 0.0001 | 0.0002 | 0.0001 |
| 16K | 0.0002 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 |
| 32K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 |
| 64K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 |
| 128K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 256K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 512K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |

Table 8: Miss ratio for *compress*, 2-way associative instruction cache

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.0007 | 0.0006 | 0.0019 | 0.0009 | 0.0017 | 0.0008 |
| 2K | 0.0006 | 0.0004 | 0.0004 | 0.0003 | 0.0003 | 0.0002 |
| 4K | 0.0004 | 0.0002 | 0.0002 | 0.0002 | 0.0002 | 0.0001 |
| 8K | 0.0002 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 |
| 16K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 32K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 64K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 128K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 256K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 512K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |

Table 9: Miss ratio for *compress*, 4-way associative instruction cache

| *Instruction* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.0007 | 0.0006 | 0.0004 | 0.0004 | 0.0003 | 0.0002 |
| 2K | 0.0006 | 0.0004 | 0.0001 | 0.0003 | 0.0003 | 0.0002 |
| 4K | 0.0003 | 0.0002 | 0.0002 | 0.0001 | 0.0002 | 0.0001 |
| 8K | 0.0002 | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0.0001 |
| 16K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 32K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 64K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 128K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 256K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |
| 512K | 0.0001 | 0.0001 | 0.0001 | 0.0001 | 0 | 0 |

Table 10: Miss ratio for *compress*, 8-way associative instruction cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.3106 | 0.2434 | 0.2920 | 0.2346 | 0.2882 | 0.2275 |
| 2K | 0.1797 | 0.1418 | 0.1837 | 0.1449 | 0.1669 | 0.1331 |
| 4K | 0.0787 | 0.0606 | 0.0859 | 0.0650 | 0.0808 | 0.0624 |
| 8K | 0.0285 | 0.0223 | 0.0331 | 0.0251 | 0.0354 | 0.0268 |
| 16K | 0.0098 | 0.0084 | 0.0087 | 0.0073 | 0.0093 | 0.0073 |
| 32K | 0.0066 | 0.0087 | 0.0049 | 0.0042 | 0.0044 | 0.0036 |
| 64K | 0.0046 | 0.0041 | 0.0030 | 0.0026 | 0.0023 | 0.0019 |
| 128K | 0.0039 | 0.0037 | 0.0023 | 0.0022 | 0.0017 | 0.0015 |
| 256K | 0.0033 | 0.0033 | 0.0017 | 0.0017 | 0.0009 | 0.0009 |
| 512K | 0.0033 | 0.0033 | 0.0017 | 0.0017 | 0.0009 | 0.0009 |

Table 11: Miss ratio for *fpppp*, 2-way associative data cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.2760 | 0.1944 | 0.2626 | 0.1870 | 0.2496 | 0.1762 |
| 2K | 0.1653 | 0.1126 | 0.1634 | 0.1158 | 0.1473 | 0.1029 |
| 4K | 0.0676 | 0.0410 | 0.0740 | 0.0454 | 0.0680 | 0.0440 |
| 8K | 0.0161 | 0.0114 | 0.0214 | 0.0135 | 0.0243 | 0.0148 |
| 16K | 0.0074 | 0.0061 | 0.0060 | 0.0046 | 0.0059 | 0.0042 |
| 32K | 0.0054 | 0.0044 | 0.0035 | 0.0028 | 0.0026 | 0.0020 |
| 64K | 0.0038 | 0.0035 | 0.0021 | 0.0019 | 0.0013 | 0.0012 |
| 128K | 0.0033 | 0.0033 | 0.0017 | 0.0017 | 0.0009 | 0.0009 |
| 256K | 0.0033 | 0.0033 | 0.0017 | 0.0017 | 0.0009 | 0.0009 |
| 512K | 0.0033 | 0.0033 | 0.0017 | 0.0017 | 0.0009 | 0.0009 |

Table 12: Miss ratio for *fpppp*, 4-way associative data cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.2743 | 0.1799 | 0.2624 | 0.1721 | 0.2509 | 0.1607 |
| 2K | 0.1502 | 0.0957 | 0.1507 | 0.0986 | 0.1363 | 0.0867 |
| 4K | 0.0651 | 0.0324 | 0.0723 | 0.0354 | 0.0648 | 0.0337 |
| 8K | 0.0126 | 0.0086 | 0.0145 | 0.0085 | 0.0189 | 0.0100 |
| 16K | 0.0072 | 0.0056 | 0.0053 | 0.0040 | 0.0047 | 0.0033 |
| 32K | 0.0053 | 0.0041 | 0.0035 | 0.0025 | 0.0026 | 0.0017 |
| 64K | 0.0036 | 0.0033 | 0.0020 | 0.0017 | 0.0012 | 0.0009 |
| 128K | 0.0033 | 0.0033 | 0.0017 | 0.0017 | 0.0009 | 0.0009 |
| 256K | 0.0033 | 0.0033 | 0.0017 | 0.0017 | 0.0009 | 0.0009 |
| 512K | 0.0033 | 0.0033 | 0.0017 | 0.0017 | 0.0009 | 0.0009 |

Table 13: Miss ratio for *fpppp*, 8-way associative data cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.2114 | 0.1729 | 0.2507 | 0.1983 | 0.3039 | 0.2376 |
| 2K | 0.1357 | 0.1136 | 0.1500 | 0.1206 | 0.1832 | 0.1446 |
| 4K | 0.0917 | 0.0784 | 0.0887 | 0.0727 | 0.1037 | 0.0827 |
| 8K | 0.0643 | 0.0558 | 0.0518 | 0.0435 | 0.0541 | 0.0436 |
| 16K | 0.0487 | 0.0417 | 0.0337 | 0.0286 | 0.0271 | 0.0224 |
| 32K | 0.0371 | 0.0300 | 0.0241 | 0.0196 | 0.0167 | 0.0136 |
| 64K | 0.0237 | 0.0200 | 0.0153 | 0.0125 | 0.0096 | 0.0078 |
| 128K | 0.0185 | 0.0160 | 0.0112 | 0.0092 | 0.0068 | 0.0053 |
| 256K | 0.0155 | 0.0145 | 0.0083 | 0.0075 | 0.0044 | 0.0039 |
| 512K | 0.0144 | 0.0142 | 0.0074 | 0.0072 | 0.0038 | 0.0037 |

Table 14: Miss ratio for *go*, 2-way associative data cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.1860 | 0.1372 | 0.2237 | 0.1550 | 0.2785 | 0.1872 |
| 2K | 0.1229 | 0.0936 | 0.1360 | 0.0955 | 0.1634 | 0.1097 |
| 4K | 0.0788 | 0.0634 | 0.0726 | 0.0539 | 0.0818 | 0.0561 |
| 8K | 0.0567 | 0.0465 | 0.0418 | 0.0328 | 0.0372 | 0.0270 |
| 16K | 0.0442 | 0.0353 | 0.0284 | 0.0227 | 0.0196 | 0.0153 |
| 32K | 0.0371 | 0.0260 | 0.0225 | 0.0162 | 0.0136 | 0.0100 |
| 64K | 0.0236 | 0.0179 | 0.0151 | 0.0109 | 0.0093 | 0.0066 |
| 128K | 0.0167 | 0.0144 | 0.0095 | 0.0076 | 0.0057 | 0.0042 |
| 256K | 0.0152 | 0.0141 | 0.0081 | 0.0071 | 0.0043 | 0.0037 |
| 512K | 0.0141 | 0.0141 | 0.0071 | 0.0071 | 0.0036 | 0.0036 |

Table 15: Miss ratio for *go*, 8-way associative data cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.1757 | 0.1226 | 0.2103 | 0.1354 | 0.2603 | 0.1622 |
| 2K | 0.1123 | 0.0826 | 0.1243 | 0.0817 | 0.1492 | 0.0927 |
| 4K | 0.0751 | 0.0586 | 0.0676 | 0.0474 | 0.0750 | 0.0475 |
| 8K | 0.0547 | 0.0435 | 0.0383 | 0.0294 | 0.0318 | 0.0223 |
| 16K | 0.0415 | 0.0328 | 0.0263 | 0.0206 | 0.0174 | 0.0134 |
| 32K | 0.0360 | 0.0247 | 0.0213 | 0.0152 | 0.0124 | 0.0091 |
| 64K | 0.0239 | 0.0174 | 0.0152 | 0.0105 | 0.0096 | 0.0064 |
| 128K | 0.0164 | 0.0141 | 0.0094 | 0.0073 | 0.0058 | 0.0040 |
| 256K | 0.0150 | 0.0141 | 0.0080 | 0.0071 | 0.0042 | 0.0036 |
| 512K | 0.0141 | 0.0141 | 0.0071 | 0.0071 | 0.0036 | 0.0036 |

Table 16: Miss ratio for *go*, 8-way associative data cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.1499 | 0.1347 | 0.1049 | 0.1210 | 0.1530 | 0.1339 |
| 2K | 0.1331 | 0.1199 | 0.1103 | 0.0976 | 0.1039 | 0.0929 |
| 4K | 0.1175 | 0.1054 | 0.0902 | 0.0794 | 0.0772 | 0.0667 |
| 8K | 0.1027 | 0.0915 | 0.0753 | 0.0654 | 0.0617 | 0.0525 |
| 16K | 0.0858 | 0.0771 | 0.0597 | 0.0514 | 0.0471 | 0.0397 |
| 32K | 0.0728 | 0.0650 | 0.0470 | 0.0408 | 0.0347 | 0.0291 |
| 64K | 0.0646 | 0.0551 | 0.0383 | 0.0322 | 0.0252 | 0.0208 |
| 128K | 0.0550 | 0.0449 | 0.0299 | 0.0243 | 0.0172 | 0.0140 |
| 256K | 0.0382 | 0.0289 | 0.0205 | 0.0157 | 0.0111 | 0.0087 |
| 512K | 0.0190 | 0.0183 | 0.0101 | 0.0095 | 0.0053 | 0.0050 |

Table 17: Miss ratio for *compress*, 2-way associative data cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.1373 | 0.1197 | 0.1180 | 0.0977 | 0.1235 | 0.0938 |
| 2K | 0.1262 | 0.1084 | 0.0994 | 0.0832 | 0.0904 | 0.0726 |
| 4K | 0.1160 | 0.0977 | 0.0881 | 0.0716 | 0.0738 | 0.0586 |
| 8K | 0.1014 | 0.0846 | 0.0742 | 0.0591 | 0.0603 | 0.0463 |
| 16K | 0.0849 | 0.0719 | 0.0591 | 0.0472 | 0.0463 | 0.0351 |
| 32K | 0.0712 | 0.0602 | 0.0453 | 0.0365 | 0.0330 | 0.0249 |
| 64K | 0.0626 | 0.0504 | 0.0363 | 0.0284 | 0.0232 | 0.0174 |
| 128K | 0.0563 | 0.0403 | 0.0304 | 0.0217 | 0.0171 | 0.0122 |
| 256K | 0.0446 | 0.0264 | 0.0232 | 0.0142 | 0.0121 | 0.0077 |
| 512K | 0.0175 | 0.0175 | 0.0090 | 0.0090 | 0.0046 | 0.0046 |

Table 18: Miss ratio for *compress*, 4-way associative data cache

| *Data* | | | | | | |
|---|---|---|---|---|---|---|
| Size | Block Size | | | | | |
| | 16 | | 32 | | 64 | |
| | LRU | OPT | LRU | OPT | LRU | OPT |
| 1K | 0.1354 | 0.1144 | 0.1103 | 0.0906 | 0.1072 | 0.0821 |
| 2K | 0.1241 | 0.1025 | 0.0972 | 0.0776 | 0.0848 | 0.0657 |
| 4K | 0.1131 | 0.0920 | 0.0857 | 0.0667 | 0.0718 | 0.0540 |
| 8K | 0.1009 | 0.0859 | 0.0738 | 0.0558 | 0.0598 | 0.0432 |
| 16K | 0.0843 | 0.0689 | 0.0585 | 0.0444 | 0.0458 | 0.0324 |
| 32K | 0.0705 | 0.0580 | 0.0448 | 0.0346 | 0.0325 | 0.0231 |
| 64K | 0.0623 | 0.0485 | 0.0357 | 0.0268 | 0.0223 | 0.0160 |
| 128K | 0.0567 | 0.0386 | 0.0303 | 0.0204 | 0.0167 | 0.0112 |
| 256K | 0.0463 | 0.0256 | 0.0239 | 0.0136 | 0.0125 | 0.0073 |
| 512K | 0.0175 | 0.0175 | 0.0090 | 0.0090 | 0.0046 | 0.0046 |

Table 19: Miss ratio for *compress*, 8-way associative data cache