The Impact of Prefetching and Victim Caching on Computer Systems Performance

Walter W. Schilling, Jr.
Multimedia Systems Department
Visteon Automotive Systems
Allen Park, Michigan 48101
walter.schilling@computer.org

Mansoor Alam
EECS Department
University of Toledo
Toledo, Ohio 43615
malam@eecs.utoledo.edu

Abstract

The performance of computer systems is heavily dependent upon the caching systems used. This article investigates the effect of two well known caching systems, Victim caching and Prefetching. Markov Models for memory systems are presented for the performance analysis of systems using the combination of these two techniques. Simulation results are presented and compared with standard caching methods. The techniques are then shown to be effective at improving computer systems performance.

1 Introduction

Improving cache performance has been the subject of hundreds of research projects. Many methods have been shown to be effective at reducing cache miss rates. To continue improving performance, it is essential that the impact of multiple methods in tandem be investigated.

Cache misses are typically modeled into three different categories, *compulsory misses*, *capacity misses*, and *conflict misses*.[3] Compulsory misses, also referred to as cold start misses or first reference misses, are misses that occur when a cache block is first read from memory. For a long running program, the compulsory cache miss ratio is usually small and can usually be ignored. Conflict misses, also referred to as collision misses or interference misses, are misses caused by two blocks mapping to the same location. Conflict misses only occur in a direct mapped or set associative cache, and typically account for between 20% and 40% of all misses in a direct mapped cache.[2] Capacity misses are those caused when the cache capacity is reached and blocks must be discarded.

Improving the cache performance can be accomplished in several different manners. Changing the block size and associativity represent scaling of the cache design. Victim Caching and Prefetching represent architectural changes to the cache.

In a victim cache, a small fully associative cache is placed between the cache and the next lower level in the memory hierarchy. The *victim cache* contains only items which have been discarded from the main cache. Victim caches are effective at decreasing thrashing of the cache, especially for direct mapped caches.

In prefetching, the goal is to "guess" what memory location the processor will request next and to start a prefetch operation in advance. Prefetching can be implemented through software and hardware. Software based prefetching relies upon the compiler placing fetch instructions in advance of the memory fetch. Hardware based prefetching looks for patterns in the address stream, which then are used to predict future memory accesses.

2 Cache Modeling

Modeling of cache systems is an important step in the design of modern computer systems. A typical processor and memory consists of a requester and server. The Requester represents the processor requesting information from the memory hierarchy, and the server represents the memory hierarchy supplying the data to the processor. Both processes are assumed to be independent of each other. The mean arrival rate for service requests, designated as λ, is measured in requests per unit of time. The mean service rate distribution is referred to using the symbol μ.

There are many standard Markov Models. However, to effectively model the memory system in terms of cache miss ratios and miss penalties, it is necessary to develop a non-standard Markov Model, shown in Figure 1. In this model, assuming that the system has

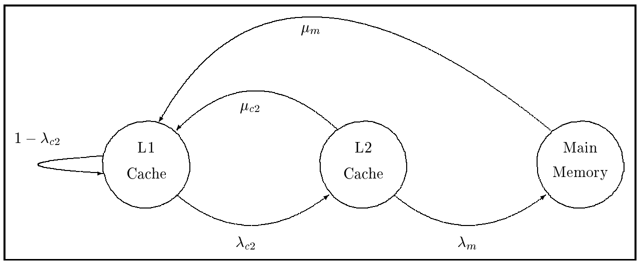

Figure 1: A simple model for the memory system interface.

just fetched from Level 1 Cache, it can fetch the next item from the Level 1 Cache, the Level 2 Cache, or the Main Memory.

The Markov Model can be classified in several different categories. Since the return time is not infinite and all states are returned to with a probability of 1, the system is *recurrent nonnull.* Because the Level 1 Cache state has a self loop, the system is *aperiodic.* The chain is irreducible, as all states are reachable from all other states. Because the chain is irreducible, recurrent nonnull, and aperiodic, it is also an *ergodic* chain. Thus, there must exist a unique limiting distribution for the probability of being in each state k which is independent of the initial state. This state is referred to as the *steady-state* or *equilibrium* state.

To determine the steady-state performance for this system, we must first determine the state transition probability matrix. This matrix relates the probability p_{ij} as the probability that the system will transition from state i to state j in one single step. For this model, the transition matrix can be represented as

$$T_m = \begin{pmatrix} (1-\lambda_{c2})\Delta t & \lambda_{c2}\Delta t & 0 \\ \mu_{c2}\Delta t & (1-\mu_{c2}-\lambda_m)\Delta t & \lambda_m \Delta t \\ \mu_m \Delta t & 0 & (1-\mu_m)\Delta t \end{pmatrix} \quad (1)$$

To determine the steady state system performance, the system is arbitrarily started in one state. After one state transition, the probability of the system being in any given state is defined as (assuming $\Delta t = 1$):

$$p(0) \cdot P = p(1) \quad (2)$$

Since all values for the system are less than one, consecutive multiplications will eventually converge to steady state values. Thus,

$$\lim_{n\to\infty} p(n) \cdot P = \lim_{n\to\infty} p(n+1) \quad (3)$$

To obtain this value, the matrix equation:

$$[\pi_1 \ \pi_2 \ \pi_3] \cdot P = [\pi_1 \ \pi_2 \ \pi_3] \quad (4)$$

must be solved. If the values for μ and λ are given, solving for the steady state values of π is simply reduced to solving a set of simultaneous equations.

To effectively model a system with prefetching and victim caching, the model is changed slightly, as is shown in Figure 2.6. An additional state is placed between the Level 1 Cache and the Level 2 Cache which corresponds to the Victim Cache / Prefetch Buffer. This model can then be solved using the same method.

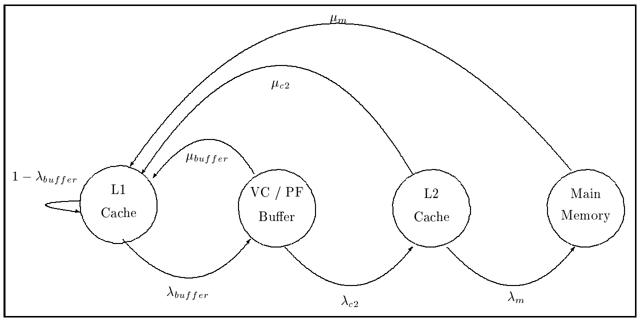

Figure 2: Memory model in which Victim Caching and Prefetching are included.

3 Prefetching and Victim Caching

3.1 Victim Caching

Victim caching is a slightly modified implementation of miss caching. Victim caching, first proposed by Norman Jouppi [2], improves on miss caching by loading the cache with memory contents that are discarded from the primary cache (the *victim* line).

In a typical victim cache implementation, when a memory fetch occurs, the address for the fetch is sent to both the victim cache and the tag memory. If a hit occurs within the primary cache, then the standard cache hit procedure is followed. However, if a hit does not occur in the primary cache, the victim cache is already being searched. If a hit occurs, on the second clock, the entry from the victim cache is swapped with the entry in the primary cache. Thus, instead of a lengthy, multi-cycle penalty, only a single cycle penalty is incurred.

If an item is not found in the victim cache, a normal fetch to memory is made. The item replaced by this fetch is then moved into the victim cache.

Victim caching always offers a performance improvement over miss caching. In general, the most dramatic results are found in systems which have a small, direct mapped cache. As cache size is increased, the victim cache necessary size to have a noticeable improvement in cache performance is decreased. As the total size of the primary cache is increased, a larger percentage of misses are caused by conflict and compulsory misses relative to capacity misses.

3.2 Prefetching

In its simplest, the concept of prefetching is to move the data to the processor in advance of when it is needed. This avoids the processor waiting for memory accesses.

Hardware based prefetching schemes fall into essentially three different categories, *spatial*, *temporal*, and *stride detecting*. In spatial prefetching, a prefetch occurs when there is a miss in the cache. In such schemes, a reference to memory block i automatically starts a prefetch to memory block $i + 1$. In temporal prefetching, data prefetching is handled by having a "lookahead" program counter which ideally stays one memory latency ahead of the actual program counter.[1]

Stride based prefetching relies upon the detection of regular, recurrent memory access patterns in the execution of the program. Standard caches are effective if the memory access pattern fits into the *scalar* and *zero stride* categories. If a cache has a large block size and the stride is small, caches can help in the *constant stride* category. The goal of any prefetching system is to generate prefetch requests for memory patterns which are detected to be in the *scalar*, *zero stride*, or *constant stride* categories. At the same time, the algorithm should attempt to avoid prefetches for memory patterns which fall into the *irregular* pattern.[1]

The basic reference prediction table prefetching scheme modified the standard instruction and data caches to include a *reference prediction table*. The Reference Prediction Table keeps track of the information required to successfully detect regular memory access patterns. The RPT is referenced to the address of the given load instruction. Each entry contains a tag, a previous address, a stride, and a state. The tag is the address of the instruction for which the entry refers. The previous address indicates the address which was last referenced when the processor executed the instruction. The stride represents the difference between the last address which was referenced and the previous reference from that instruction. The state is a set of encodings which designate the regularity of the memory accesses, including *initial*, *transient*, *steady*, or *no prediction (null)*.

4 Prefetching and Victim Caching Combined

There are many methods for combining prefetching and victim caching into a single system. In the simplest implementation of victim caching and prefetching, one would create two independent structures. One structure would house the prefetch buffer and reference prediction tables. The second structure would hold the victim cache. This method, however, has problems. First and foremost, in checking for a hit, there are three locations to search. This adds extra complexity to the search mechanism. Secondly, this adds significantly to the transistor count on chip.

To remedy these problems, the architecture of Figure 3 has been proposed. In this architecture, the victim cache and prefetch buffers are combined into a single mechanism. This reduces the search complexity, for only two areas need to be searched simultaneously.

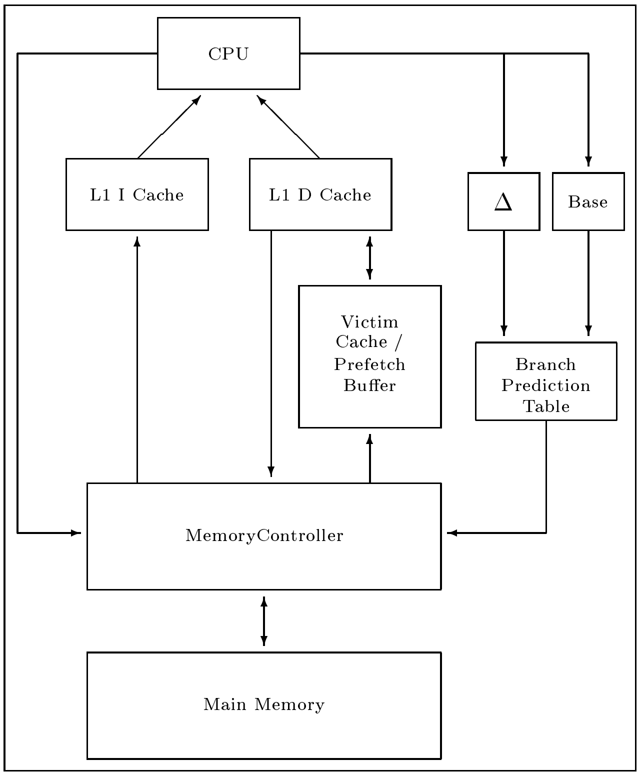

Figure 3: The Architecture for the system.

Typical operation of the chip is as follows. Under normal operation, the Reference prediction table will detect repetitive patterns in memory access. Whenever such patterns exist, the memory controller will compare the address of the next prefetch with those in the cache and buffer. If a match exists, nothing happens. However, if the item is not present in either the victim cache / prefetch buffer or primary cache, a prefetch request is sent to the main memory. Items which are discarded from the data cache are sent to two places, the main memory and the victim cache.

Cache misses can generally be divided into three categories. A system which implements prefetching will remove a portion of cache misses. A system which implements victim caching will also remove a portion of the cache misses. However, since these sets are not mutually exclusive, there will be some classes of cache misses that both prefetching and victim caching remove. Thus, the total reduction in cache misses must be expressed as:

$$Cache\ Miss\ Reduction = VC + PF - VC \cap PF \quad (5)$$

where:
VC = cache misses removed through victim caching
PF = cache misses removed through prefetching
$VC \cap PF$ represents cache misses which are removed by both prefetching and victim caching.

Once strides and base offsets have been established, the prefetching mechanism should remove most compulsory misses. The victim cache implementation should effectively reduce capacity misses, as well as conflict misses within the cache. Thus, under ideal circumstances, the only cache misses which should occur are compulsory misses that occur before the reference prediction table has entered steady-state operation.

5 Memory Simulator Testing and Results

To test combination of victim caching and prefetching, a memory simulator was developed. This simulator simulated the desired memory architecture through address traces. Three different address traces, namely CC, Tex, and Spice, from the SPEC benchmark suite were simulated.

The first simulations were designed to verify whether or not the simulator was functioning properly as well as providing a starting point for comparing the results of the other tests.

From cache theory, it is known that two general trends occur when varying the cache size. First, as one increases the overall size of the cache while keeping the block size fixed, the miss ratio generally decreases. Second, as one increases the block size, the miss ratio generally decreases.

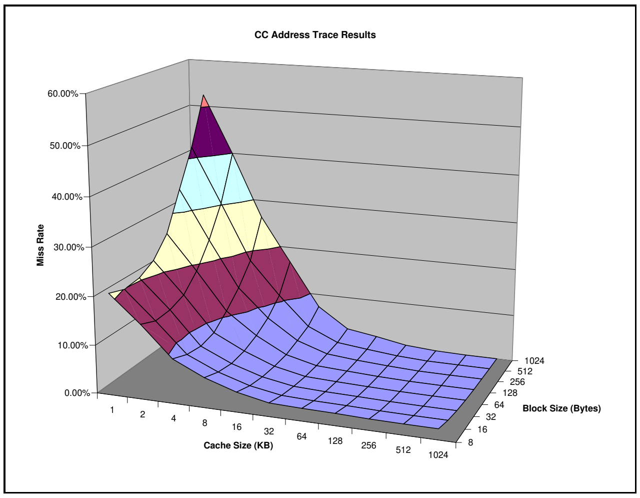

Figure 4: Cache simulations for a standard cache using the CC address trace.

It was expected that plotting the total cache size on one axis, the block size on the second axis, and the miss rate on the third axis, a three-dimensional figure would be created which generally decreases from a high miss rate when the total cache size is small and the block size is large to a low miss rate when the cache size is large and the block size is small. In general, the simulator has verified this. In all three address traces tested (CC, Spice, and Tex), this trend was exhibited.

Once it was determined that the standard cache was functioning properly, the victim cache was enabled. The simulator returned four pieces of significant data, including the number of cache searches performed, the number of cache misses, the number of victim cache searches, and the number of hits in the victim cache.

The victim cache was searched every time there is a miss in the main cache. Thus, the number of victim cache searches was equal to the number of main cache misses. Since the victim cache could be accessed with only a single cycle penalty (as opposed to the approximately 10 cycle penalty for accessing the main memory), the statistic which was desired was the miss ratio for the main cache and victim cache. Numerically, this can be expressed as:

$$M.R. = \frac{Main\ Cache\ Misses - Victim\ Cache\ Hits}{Total\ Memory\ Accesses} \quad (6)$$

To best express the performance improvement, one has to compare the miss ratio with victim caching (and the various sizes of the victim cache) with those of the system when the victim cache is disabled. In all cases, no degradation was experienced by adding a victim

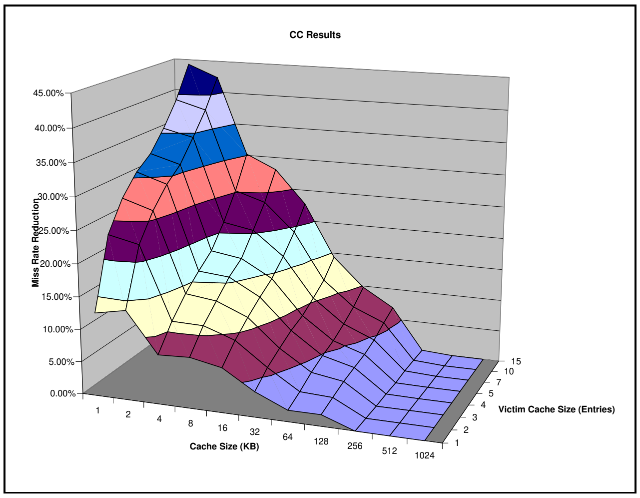

Figure 5: Victim Cache simulations using the CC address trace. Block size is fixed.

cache and as the size of the victim cache was increased, the performance improved.

Once the simulations with the victim cache were completed, the victim caching mechanism was disabled and the prefetching mechanism was enabled.

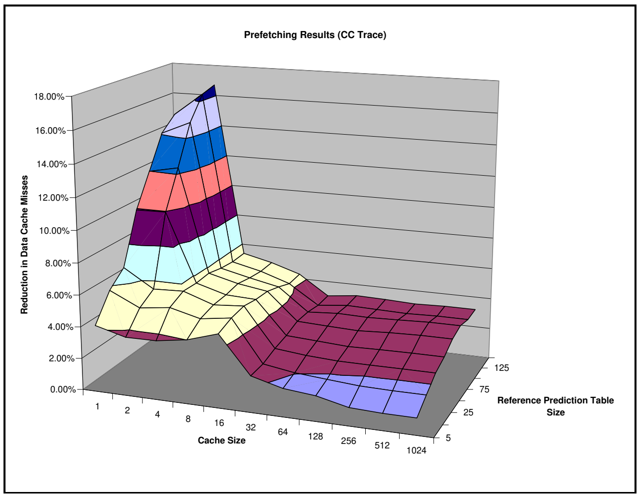

Figure 6: Prefetching Cache simulations using the CC address trace. Block size is fixed. Prefetch buffer size is fixed. Prefetch delay is fixed.

As was done with the Victim Caching results, the results for prefetching were compared to the standard cache example in terms of percent reduction in misses.

The largest improvement in performance occurred with smaller cache sizes and large reference prediction tables. As the size of the reference prediction table was decreased, the performance improvement in terms of reduction in cache misses decreased. Additionally, as the cache size increased, the percentage improvement from the prefetching mechanism decreased.

6 Victim Caching and Prefetching Results

The final simulations involved enabling both the prefetching and victim caching mechanisms. Operating in this mode, items which were discarded from the main cache were placed in the victim cache / prefetch buffer. The prefetching mechanism also attempted to prefetch memory references into the victim cache / prefetch buffer.

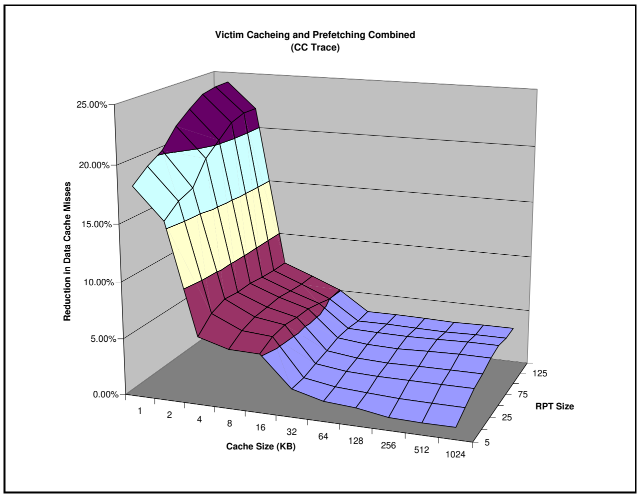

Figure 7: Victim Caching and Prefetching enabled simulations using the CC address trace. Block size is fixed. Prefetch buffer size is fixed. Prefetch delay is fixed.

In the simulation results, two general trends were noticed. First, when varying the size of the reference prediction table, increasing the size of the reference prediction table generally increased the performance improvement. This is understandable, for increasing the number of reference prediction table entries increases the number of load and store instructions which can be tracked by the mechanism. If more load instructions can be tracked, there is a greater probability that the current instruction will be traced and in the steady state. Second, as the size of the Victim Cache / Prefetch Buffer was increased, the performance improvement also increased. This trend is also logical, for as the size of the Victim Cache / Prefetch Buffer is increased, there is a greater likelihood that a given entry will not be forced from the buffer. Both of these trends are more noticble as the cache itself is smaller. This is most likely caused by the fact that a

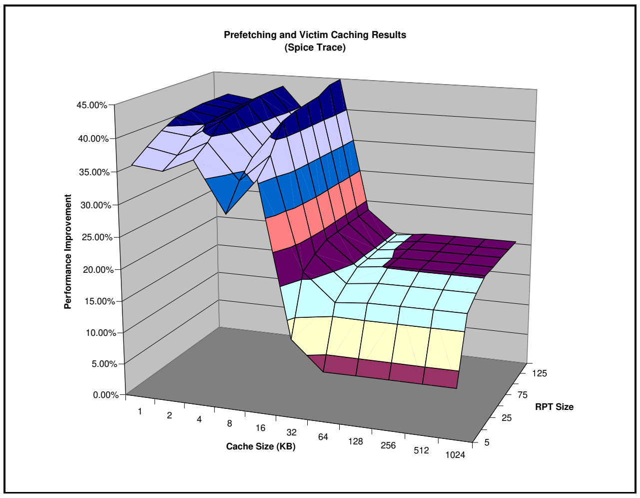

Figure 8: Victim Caching and Prefetching enabled simulations using the Spice address trace.

small cache in general has more misses than a larger cache (assuming a fixed block size), and thus, there is more room for performance improvement.

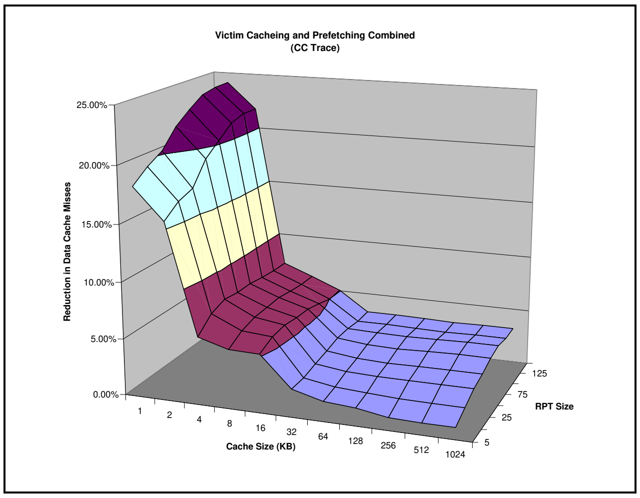

Figure 9: Victim Caching and Prefetching enabled simulations using the Tex address trace.

The net improvement when victim caching and prefetching are combined varies widely. However, there are some general trends which occur in the data.

In all cases, when the primary cache is at its smallest size, the net improvement for using Victim caching and Prefetching is less than the value when only Victim Caching is used. This indicates that erroneous prefetches are increasing the miss rate by removing valid victim caches from the buffer. Near the middle of the cache size range, there is an area where the reduction in cache misses is greater than from either method. This indicates that in this range, the combination of victim caching and prefetching is functioning as designed, yielding a net performance improvement. As the size is further increased, the gain from Victim Caching approaches zero, and all of the performance gain is obtained from the Prefetching mechanism.

Each change in performance corresponds to a change in the type of cache miss. With a small cache, there are many conflict and capacity misses. As the cache size is increased, these misses begin to diminish, and thus, the victim cache is less effective at removing the Cache misses. When the cache is larger, primarily the only cache misses remaining are compulsory misses. This is why the prefetching mechanism is effective for the larger cache sizes. The conflict and capacity misses have been effectively removed by increasing the cache size.

7 Summary

The combination of Victim Caching Prefetching has been shown to be effective at reducing cache miss rates. With small sized caches, the victim cache helps to reduce conflict and capacity misses. With larger size caches, the prefteching mechanism helps to reduce compulsory misses.

Acknowledgements

This research was supported by a grant from the Ohio Supercomputing Center, located in Columbus, Ohio.

References

[1] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Electronic Computers*, 44(5):609–623, 1995.

[2] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings 17th International Symposium on Computer Architecture*, pages 364–373, Los Alamitos, 1990. IEEE Computer Society Press.

[3] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware Software Interface.* Morgan Kaufmann Publishers, Inc., San Francisco, 1994.