Cluster Miss Prediction for Instruction Caches in Embedded Networking Applications

Ken Batcher
Kent State University, Cisco Systems
4125 Highlander Parkway
Richfield, OH 44286
330-523-2044
batcher@cisco.com

Robert Walker
Kent State University
Computer Science Department
Kent, OH 44242
330-672-9105
walker@cs.kent.edu

ABSTRACT
In this paper, we describe a new method of instruction prefetching that reduces the cache miss penalty by anticipating the cache behavior based on previous execution. Our observations indicate that instruction cache misses often repeat in *clusters* under certain conditions prevalent in real time embedded networking systems. By identifying the start of a cluster miss sequence and preparing an instruction buffer for the upcoming cache misses, the miss penalty can be reduced if a miss does occur. A sample industrial networking example is used to illustrate the effectiveness of this technique compared with other prefetch methods.

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – cache memories; C.1.2 [**Processor Architectures];** C.4[**Performance of Systems].**

General Terms
Performance, Design, Experimentation.

Keywords
Cache design, embedded systems, networking, cache prefetch, hiding memory latency, compulsory cache misses, WCET.

1. INTRODUCTION
In modern embedded systems, optimizing the cache performance can be crucial to achieving a marketable design. Embedded systems designers must live within tight constraints to maximize performance in a low cost system. Embedded systems usually run dedicated applications, but these dedicated applications may not be well suited to traditional cache architectures, which are designed for general purpose applications. Fortunately, embedded systems can be tuned to use smaller, more efficient cache architectures better suited for dedicated applications.

This paper focuses on one specific dedicated application ─ real-time embedded networking. This application is often data driven and reactive in nature, and must operate under real time processing constraints where the software behavior can change radically, depending on the system load and packet mix behavior at a particular instance in time [14, 20]. Features such as interrupt processing, cryptographic algorithms, and context switching make traditional cache architectures less effective for embedded networking applications.

To gain the time-to-market advantage of IP reuse, embedded systems are often implemented using synthesizable CPU cores with internal caches of modest size. These commercially available cores often form the central component in an ASIC design which is combined with custom logic to comprise an embedded system [15, 21]. Due to the small cache size, high miss rates can be quite common in such systems, meaning performance can be improved substantially if the miss penalty can be reduced.

Embedded systems are often implemented using shared internal busses to transport data between the CPU and ASIC modules [4, 22]. In a typical embedded system, a single external memory is also available which must be shared between the CPU and various ASIC modules that can perform DMA access. Unfortunately, this bus sharing adds to the overall cache miss penalty since the refill time for cache misses is adversely affected by the arbitration delay for the memory.

These unique requirements of embedded systems, particularly for networking applications, demand an efficient, low-cost cache support architecture. The technique presented here uses a Cluster Miss Prediction Buffer (CMPB) placed on the shared bus between the CPU core and the external memory (Figure 1), which attempts to reduce the cache fill penalty by predicting a *cluster of instruction misses* based on previous cache miss behavior. The CMPB also reduces the overall system bus traffic since misses are serviced by the buffer, not the external memory.

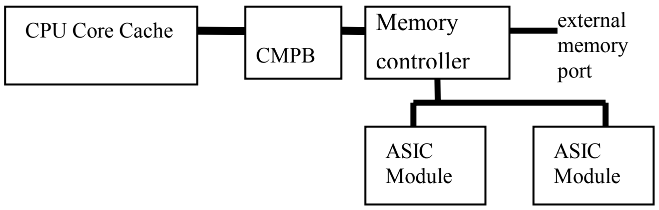

Figure 1. Embedded System with CMPB.

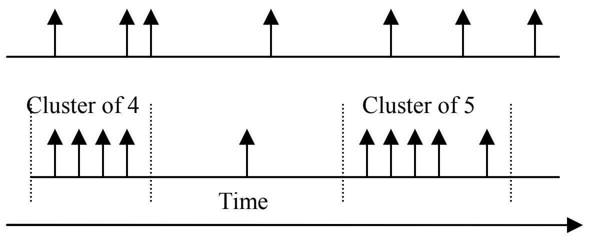

Figure 2. Cluster miss Identification.

2. Related Work

Embedded system designers often employ a variety of different techniques to improve CPU instruction cache performance. Many of these are quite effective and often are focused on solving the problem with regard to one or more of the following factors: low power, lost cost (silicon area), or high performance [2].

A common theme of many of these techniques is the inclusion of a prefetch buffer external to the CPU core and L1 cache. *Prefetching* attempts to reduce memory latency associated with a cache miss by predicting upcoming cache line accesses. If an upcoming cache line is not currently cached in the CPU, a good prefetch method can load the cache line into the cache now and prevent a miss later. A related technique also fetches the upcoming cache line from main memory but stores it into the prefetch buffer instead, commonly known as a partial miss.

A classical concept often used to service the prefetch buffer uses streaming buffers that prefetch the next sequential line on a miss [9, 11]. These techniques attempt to exploit the temporal and spatial locality of instruction accesses, since the next cache access often follows sequential execution order.

Unfortunately, most prefetch techniques are not as effective for embedded networking. First, most of these techniques suffer in the presence of program branches. While there are some prefetch techniques that handle branches using dynamic prediction [3,17], those methods can not be used effectively in embedded systems employing CPU cores since they require access to information deep in the processor core that may not be exposed outside the core (e.g., branch prediction tables, branch history tables, or pipeline state). Moreover, the unpredictable instruction flow behavior of embedded networking leads to many context switches and interrupt vectoring, again rendering prefetching less effective.

The real-time nature of embedded networking also makes prefetching difficult. As a result, real time systems use techniques such as cache line locking, column caching, software prefetching, and scratch pad memory to improve predictability for time critical portions of code [1, 16]. Methods used to quantify, improve and bound the worst case execution time (WCET) of real time systems are described in [6, 10]. These methods exploit the fixed nature of the embedded applications to tune the execution flow to the embedded memory organization, allowing improved predictability of performance in a multi-tasking environment.

3. Cluster Miss Prediction Buffer (CMPB)

As shown in Figure 1, our Cluster Miss Prediction Buffer (CMPB) is external to the CPU core, sitting on the shared bus that gives the CPU access to the external memory controller. The CMPB observes instruction cache miss activity. If a miss occurs

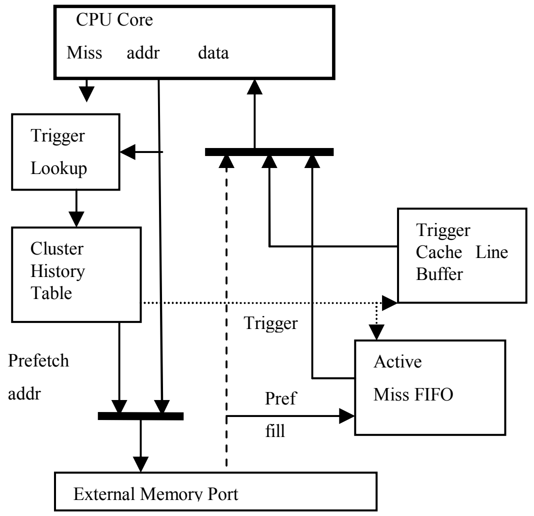

Figure 3. CMPB Architecture.

and is correctly predicted, the CMPB will intercept the memory access and provide the instruction cache line, thus forgoing the memory access. This reduces the cache miss penalty as well as traffic on the shared memory bus. This method is similar to the traditional prefetch on miss technique, but this prefetch is selective and occurs only at the start of an identified cluster miss.

The non-intrusive nature of this design is important for practical reasons when CPU cores are used. These licensed cores are often provided as a black box, one which does not allow the designer to access the internal workings of the core for intellectual property reasons. Having the CMPB sit on the bus external to the CPU also avoids adding logic internal to the CPU core that might reduce the execution speed of the core or its internal cache.

An *instruction cluster miss* is defined as a string of strongly related instruction cache misses that occur in a relatively short period of time and are repetitive throughout the execution. An example is shown in Figure 2, where the arrows indicate instruction cache misses and the horizontal axis represents time. In the top sequence, there is not a strong occurrence of clustering since the misses are spread out in time. In contrast, the bottom example shows two cluster miss candidates ─ instances where the time between the misses is relatively short.

The CMPB concept is described in the remainder of this section and illustrated using a sample industrial networking application. The hardware consists of shared bus 32-bit embedded system (similar to Figure 1) with a MIPS CPU core, using a 2-way set associative 8KB instruction cache with 32 byte line size [5].

3.1 CMPB Prefetch Architecture

The CMPB is optimized to improve performance when servicing a burst of instruction misses. Embedded networking applications, with their real time processing constraints, experience the most difficulty under these circumstances. Unfortunately, these clusters of misses can be quite common, due to the rapid context switching and control based execution of networking applications coupled with the small caches of embedded systems. Moreover, the instruction fetch behavior depends on the incoming flow of network packets, which cannot be anticipated and makes prediction difficult. However, an inopportune string of cache misses might cause a packet to be dropped if it takes too long to service the cache from the external memory, a situation which we would prefer to avoid if possible.

Many of the cache misses that occur in these situations are similar to compulsory misses [19]. *Compulsory misses* are those performed when a program is first loaded, when a program is reloaded due to a context switch, or when a procedure call is made to instructions that were once in the cache but have since been flushed.

To provide the necessary cache support in these embedded networking applications, the CMPB must meet several key objectives. First, it must find the best cluster misses (those that provide the best performance improvement) to retain in a history buffer based on previous CPU miss behavior. Then it must service those instruction cache misses by delivering the data to the CPU core, thus hiding the latency of external access from the CPU. It must also be accurate in its prefetching, to avoid increasing traffic on the shared bus. Finally, it must be small in terms of silicon area. Our CMPB design, shown in Figure 3, meets these objectives as we will demonstrate.

3.2 CMPB Cache Miss Servicing

To identify and service a cluster miss sequence, the CMPB uses a *Cluster History Table* (see Figures 3 and 4). The CMPB is indexed associatively by a *trigger address* corresponding to the start of a cluster of cache misses. Each entry in the Cluster History Table then contains the next addresses in the cluster.

The CMPB services cache misses as follows. When an instruction cache miss occurs, a trigger address lookup is done to see if a matching entry exists in the Cluster History Table. If the entry does exist, the *Trigger Cache Line Buffer* is referenced to service the miss. This buffer contains one cache line for each trigger address (the first miss of each cluster) in the Cluster History Table. To save memory space, the other cache lines in the cluster are not retained, but are prefetched into the *Active Miss FIFO* as soon as the trigger address is matched. Thus a trigger address miss causes the Trigger Cache Line Buffer to service the immediate miss, and causes the Active Miss FIFO to start filling in anticipation of upcoming cache misses in the cluster.

For our sample industrial networking application, each entry in the Cluster History Table (16 entries total) is held in a concise 132 bit word. Since the sample application is a 32-bit machine with a cache line size of 8 words (32 bytes), only the upper 26 bits of address are needed to address a cache line. The Cluster History Table identifies the clusters, so the Trigger Cache Line Buffer also holds 16 cache lines and results in a small associative address lookup which can be resolved in a single clock cycle.

A cluster consists of 4-6 cache misses (the trigger plus 3-5 Cluster History Table entries) but can vary with the application, as history entries can be null if the cluster is less than 6 misses. Since CMPB is a new method, ad hoc experimentation was performed to derive the cluster size which seemed to give good results and at the same time stay in the memory budget.

The Active Miss FIFO holds 4 cache lines to service the cache line fills of the cluster. This FIFO is a small dual ported memory which allows the CPU read access while the prefetch logic writes the cache line in from the external memory. In this way, the FIFO does not appreciably add to overall memory latency when cache line fills are serviced through the FIFO.

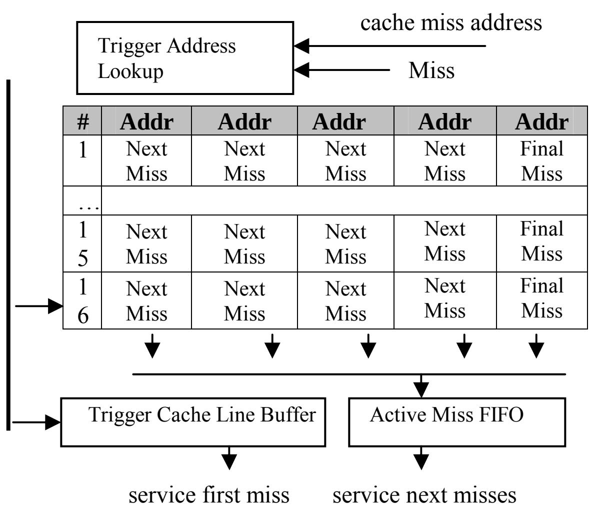

Figure 4. Cluster Miss Predication History Table Format.

For the embedded application, the goal was to stay under a system memory budget of 2K bytes allocated for cache prefetching. The total memory requirements are thus minimal and on par with other prefetch methods:

Trigger Address Lookup	= 16 entries x 4	= 64 bytes
Cluster History Table	= 16 entries x 16	= 256 bytes
Trigger Cache Line Buffer	= 16 entries x 32	= 1024 bytes

Active Miss FIFO = 4 entries x 32 (dual port equiv.) = 256 bytes

When a cluster sequence is predicted, the cache line from the Trigger Cache line Buffer is returned to the CPU, thereby saving an external memory fetch of the cache line. Furthermore, the trigger address match will cause the prefetch logic to start prefetching the next locations of the cluster from external memory. This allows full or partial overlap of the external bus cycles with CPU execution as conceptually illustrated in Figure 5.

As Figure 5 shows, without prefetching the external memory must be accessed each time a cache miss occurs. These accesses can result in significant latency, especially when external memory assess must be arbitrated with other ASIC modules. Prefetching with CMPB allows quicker miss servicing since the first trigger miss is retained in local memory and subsequent misses are prefetched into the FIFO, hiding the latency of the external access from the CPU during cache line fill. The goal of the cache miss servicing is "just in time" processing. The prefetch logic attempts to stay at least one step ahead of the CPU, returning the instruction stream for upcoming misses. Subsequent misses of a cluster are checked against the FIFO contents for servicing. The CMPB thus uses a "prefetch ahead" distance of 1-4 cache lines (up to the maximum FIFO depth).

Provided CMPB initiates prefetches ahead of the actual CPU misses with the pipelined fashion shown in Figure 5, the CMPB can improve the memory latency for the cache fill. In cases where the active miss FIFO cannot keep up or incorrectly predicts the next miss, the data can be directly forwarded to the CPU via the normal path, bypassing CMPB altogether as shown with the

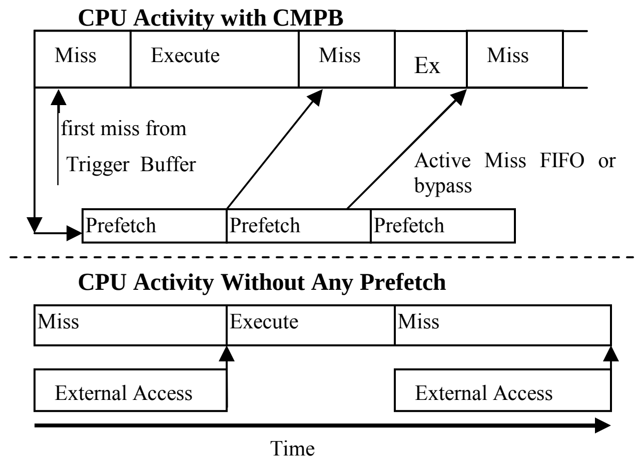

Figure 5. CPU Activity Using Cluster Miss Servicing.

dashed line in Figure 3. The dual ported nature of the active miss FIFO allows misses to be serviced as they stream in from external memory.

The CMPB is similar to prefetch on miss (POM) techniques with streaming buffers [12], but with the following notable exceptions:

- The CMPB can prefetch cache lines in any order rather than sequential prefetching (next cache line address).
- The CMPB can help the first miss of a sequence since the line is retained in the trigger buffer, while POM only helps the following misses since a miss must occur to initiate prefetch.
- The Cluster History Table contents, which determine the cache cluster miss sequence, are pre-tuned to the embedded application based on the observed behavior of the code execution, so the CMPB does not react to dynamic behavior.
- The CMPB only prefetches lines corresponding to a cluster sequence, meaning it could be referred to as "prefetch on worst miss sequence" rather than prefetch on all misses.

3.3 Classification of CMPB Cluster Misses

The CMBP relies on profiling code execution using instruction traces to determine the best cluster misses to retain in the Cluster History Table. In general, the CMPB looks for worst case scenarios where the instruction cache performs poorly executing the embedded application. Since embedded applications typically run a dedicated software program, preloading the Cluster History Table is an effective technique to maximize the performance of an embedded application. Such code profiling is popular in embedded systems using instruction caches [7, 18].

Each entry in the Cluster History Table refers to a miss address that previously was strongly associated with a subsequent string of related misses. The best miss sequences are those that are repetitive and cause the most CPU stall cycles when servicing, so the best clusters are selected by studying the cache miss behavior of a live execution trace. Statistical averaging is performed to identify on a percentage basis the worst clusters.

An important metric used by the CMBP is the number of execution cycles during a cluster of misses. If this number is relatively low, it means the CPU was stalling repeatedly when the cluster miss occurred. If this number is relatively high, it means

Table 1. Profile of Four Cluster misses in Trial Application.

#	Avg Exec Cycles Per Miss	Occurr -ences	Cluster Size	Average CML Per Miss	Norm. Cluster Ratio
1	2.6	20	4	36.2	13.9
2	4.11	31	4	35	8.5
3	3.6	19	6	31	8.6
4	6	30	5	30.4	5.1

the cluster of misses did not degrade CPU execution significantly (e.g., the cache misses are spread out). This gives a metric of the temporal locality of each cluster versus normal execution flow.

Cluster occurrences are accumulated to determine the frequency of occurrence and the detrimental execution effect of each cluster, so the benefit of retaining the cluster in the Cluster History Table can be measured. Table 1 describes four clusters in our sample networking application. In some cases the cluster size may be small, for example only the trigger plus 3 additional cache lines for a size of 4, but, the frequency of cluster occurrence and/or slow down in CPU execution when the cluster miss occurs may be very high. For this reason, the execution cycle count is *normalized* by the size of the cluster to derive an Average Execution Cycles Per Miss metric, which allows clusters to be identified which might be less than the maximum size.

The *Cache Miss Latency* (CML) metric can be calculated by examining the performance of the external memory and memory controller. When a cache miss occurs in a cluster, the number of instruction clocks (stall cycles) until the cache miss is serviced and execution resumes are counted, and then averaged over all misses to obtain an average miss latency. It often takes many cycles to fill the cache line and resume CPU operation, especially in a shared bus architecture where the external memory access also suffers from arbitration delay if other DMA units also desire access. Furthermore data cache accesses can further contend for the external memory and delays occur with bus protocol overhead.

Embedded systems commonly use low-cost SDRAM memories. Current SDRAM memory speeds (100 MHz, 133 MHz, 166 MHz) are approximately 2-3 times slower than the CPU execution speed. Typically these memories typically require 13 cycles per burst of four data read transfers, assuming SRAM open/close occurs. For our sample networking application with 32 byte cache line and 32-bit bus, it takes two bursts to fill a single line.

A cache-based CPU commonly stalls until the first portion of the cache line is retrieved, often using critical-word-first servicing. Thereafter, the remaining misses stream into the CPU via SDRAM with little or no stall penalty. In our application, the overall CML per miss averaged 17 half-speed memory cycles or 34 CPU stall cycles. Half of the delay was due to arbitration with other ASIC units on the busy external memory. The arbiter was a fair round robin where 50% of cycles were reserved for the CPU with the rest allotted for other ASIC modules.

If the cache miss can be serviced by the prefetch logic, the gain is significant. The CPU bus speed is fast, running at the CPU bus frequency. With single cycle latency to compare the miss address to the history table and a cycle of overhead to read the trigger cache line buffer, it took only 2 CPU clocks to obtain the first critical word and resume execution. The subsequent miss prefetching may result in longer stalls (see Figure 5) since the CPU may ask for the next cache line miss before it is fully retrieved from external memory, but even those partial miss stalls are preferable to the full 34 clock penalty. This was the case for our sample networking application since the CML was quite high relative to CPU execution speed.

The methodology used to create and measure the effectiveness of the CMPB history table of M hand picked entries is as follows:

1. Run the target application in a real world environment with instruction trace enabled.
2. Log each instruction cache miss and examine the number of instruction execution cycles *between* the cache misses.
3. Post-process the trace to determine the best N candidate clusters where N > M. Use statistical averaging to get the N most significant clusters, top 1% or 2% for example.
4. Record how many times each candidate cluster appears in the trace and the average time between each instruction cache miss within the cluster. Use statistical averaging weighted by frequency of occurrence and the miss penalty.
5. Select the best M out of N candidate clusters by taking the maximum values using the following formula: (Normalized Cluster Ratio x Number of Occurrences).
6. Load the history table and trigger cache line buffer at boot time with the M candidate clusters.
7. Rerun the application; see how much CMPB method helps.

The CML and execution cycles for the cluster are normalized by size of the cluster to come up with normalized figures per cache miss. This ratio (higher value indicates more clustering) indicates the overall detrimental affect on CPU performance of the cluster miss irrespective of cluster size. When multiplied by the number of occurrences, it allows ranking of the clusters to determine the best ones to hold in the history buffer thus aiding overall CPU performance (reduced number of CPU stalls).

$$\textbf{Normalized Cluster Ratio (NCR)} = \frac{\text{Average CML per Miss}}{\text{Ave Execution Cycles Per Miss}}$$

4. Experimentation

An evaluation of the CMPB method was performed by running target software on the embedded networking application and evaluating the performance via simulation. Cryptographic suites [13], network software routines, and packet traces taken from Netbench [8] are used to emulate software processing of packets. Cycle-accurate RTL hardware simulations were used to evaluate the performance of the CMPB compared to prefetch on miss techniques (POM). A prefetch distance of one and four (four is same prefetch distance as CMPB) cache lines where evaluated using the POM method. Metrics were calculated to determine the prefetch efficiency of CMPB compared to POM methods. Table 2 describes the application run in the system with the size and approximate relative execution frequency % (in CPU cycles).

This identical software program was executed and the miss behavior measured for each method to process a fixed number (100) of identical packets. The baseline test used no prefetching, thus all cache misses serviced across system bus to SDRAM taking the average 34 clock penalty.

Table 2. Sample Code Profile in Trial Application.

#	Program	Instruc Size KB	Exec %	Description
1	crc	1	10	Crc calc from netbench
2	Hmac-sha1	3	17	message signature
3	rekey	5	7	Key mixing algorithm
4	process	5	26	Process/schedule a packet from queue
5	drr	2	5	Deficit round robin from netbench
6	Table lookup	1	7	Addr matching, also ACL filtering
7	irq	13	3	CTX switch Irq handler for packets
8	Des	8	5	Crypto blk cipher
9	Other	9	20	Misc subroutines & IP packet routines

Table 3. Results of CMPB Simulations.

#	Method	Total Cycles to Service Misses	% Improve	Prefetch Efficiency
1	POM1	346,287	14.4	79%
2	POM4	343,501	15.1	72%
3	CMPB	365,001	9.78	100%
4	Baseline	404,600	--	---

Each simulation experienced the same 11,900 cache misses during the packet processing. Table 3 shows the improvement of the total CPU cycles it took to service the cache misses compared to the baseline. The prefetch efficiency relates to the accuracy that the technique correctly identified the prefetch as the next miss [11]. Thus out of the 11,900 misses, POM1 resulted in 2499 wasted cycles and POM4 resulted in 3332 wasted prefetches. CMPB method resulted in only useful prefetches, but still a decent performance without the wasted bus bandwidth activity.

For CMPB mode, sixteen clusters were identified using the methods described in Section 3.3. The clusters often occurred when the cache was reacting to a change in execution behavior, such as that caused by crossing program boundaries, invoking a distant sub-routines, or handling a context switch in response to an interrupt.

Eight of the cluster misses exhibited poor sequential but high temporal locality. The remaining 8 clusters showed sequential and temporal locality indicating that they could have been picked up by the POM method. However in cases where POM failed, the instruction flow was greatly disturbed. This was to be expected since the inaccurate prefetch(es) from external memory must finish before the correct data is retrieved, aggravating the miss penalty and thereby resulting in higher WCET.

For POM, a miss resulted in an average of 49 cycles instead of the 34 cycles. For POM4 the average was 55 cycles since commonly 1-2 full cache lines where fetched before aborting the sequence with the correct prefetch. In some cases the missed prefetch activity did not harm the instruction flow since the time between misses was so large, but still contributed to wasted bus activity. Since CMPB works in absence of sequential behavior, it helps the most in situations where POM fails and where bus efficiency is most important.

5. Conclusion and Future Work

In the context of embedded networking applications, we have described a new method of instruction prefetching that reduces the cache miss penalty. We have shown that instruction cache misses often occur in *clusters* under certain conditions prevalent in real time embedded networking systems, and that by identifying the start of a cluster miss sequence and filling an instruction buffer for the upcoming cache misses, the miss penalty can be reduced if a miss does occur. Finally, we demonstrated this method using a sample industrial networking example to compare its effectiveness compared to the POM prefetch method.

A key observation is that our CMPB method is highly efficient. For the sample application, the selected clusters resulted in no inaccurate prefetches and hence did not contribute to wasted bus bandwidth. CMPB was highly selective in terms of what it chose to prefetch as compared to the POM techniques. These results show that there is a tradeoff between aggressive prefetching vs. selective prefetching with regard to performance, where CMPB gives decent performance boost with much less prefetch activity.

This observation leads us to speculate that the CMPB technique could be used to compliment POM technique, with clusters selected only in cases where POM would fail to predict the cache miss sequence (e.g. code that does not execute sequentially). For example, if a cache miss is not part of an identified CMPB cluster, it could degenerate to POM to service the Active Miss FIFO (Figure 4) rather than CMPB. Current experimentation using fully randomized packet mixes has shown CMPB combined with POM to be a promising technique with regards to dynamic behavior. Additional experimentation will also attempt to quantity the improvement in WCET when using CMPB method.

6. REFERENCES

[1] Choiu, D., Jain, P., Rudolph, L., Devadas, S. Application Specific Memory Management for Embedded Systems Using Software-Controlled Caches. Proceedings of 37th Design Automation Conference, pp 416-420, June 2000.

[2] Panda, P.R., Dutt, N., and Nicolau, A. *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*. Kluwer Academic Publishers, 1999.

[3] Reinmann, G., Calder, B., Austin, T. Fetch Directed Instruction Prefetching. Proc. of the 32nd Annual Int. Symposium on Micro Architecture, Nov. 1999, pp. 16-27.

[4] Lysecky, R., Vahid, F., Prefetching for Improved Bus Wrapper Performance in Cores. ACM Trans. on Design Automation of Electronic Systems, vol 7, no.1, Jan. 2002.

[5] Hamblen, J., A VHDL Synthesis Model of the MIPS Processor for Use in Computer Architecture Laboratories. IEEE Trans. On Education, vol 40, no 4, Nov 1997.

[6] Sebek, F. Instruction Cache Memory Issues in Real-Time Systems. PHD Thesis, Malardalen Univ., Sweden. 2002.

[7] Cotterell, S., Vahid, F., Synthesis of Customized Loop Caches for Core-Based Embedded Systems. Int. Conf. on Computer Aided Design, Nov. 2002, pp. 655-662.

[8] Memik, G., Mangione-Smith, W., Hu, W. Netbench: A Benchmarking Suite for Network Processors. Proc. of Int. Conference on Computer-Aided Design. Nov. 2001.

[9] Smith, A.J. Cache Memories. Computing Surveys, 14, September 1982.

[10] Tan, Y., Mooney, V., A Prioritized cache for Real-Time Systems. Proc of the 11th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI'03), pp. 168-175, April 2003.

[11] Hsu, W., Smith, J., A Performance Study of Instruction Cache Prefetch Methods. IEEE Transactions on Computers. May 1998, Vol. 47, No. 5. pp. 497-508.

[12] Jouppi, N., Improving Direct Mapped Cache Performance by the Addition of Small Fully-Associative Cache and Prefetch Buffers. 17th International Symposium on Computer Architecture, June 1990. pp 364-373.

[13] Ravi, S., et al. System Design Methodologies for Wireless Security Processing Platform. Proc. of 20th DAC June 2002. pp 777-782.

[14] Wuytack, S., Silva, J., Catthoor, F., Jong, G., Ykman-Couvreur, C. Memory Management for Embedded Network Applications. IEEE Trans. On CAD Design of Integrated Circuits and Systems, vol 18, no 5, May 1999.

[15] Crosbie, N., Kandemir, M., Kolcu, I., Ramanujam, J., Choudhary, A. Strategies for Improving Data Locality in Embedded Applications. Proceedings of 15th Int. Conf. on VLSI Design, 2002.

[16] Benakar, R., et al. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. 10th International Workshop on Hardware/Software Codesign, May 2002. pp 73 – 78.

[17] Chiu, J., Shiu, R., Chi, S., Chung, C. Instruction Cache Prefetching Directed by Branch Prediction. IEEE Proc., Computers & Digital Techniques, vol 146, no. 5, 1999.

[18] Gordon-Ross, A., Cotterell, S., Vahid, F. Exploting Fixed Programs in Embedded Systems: A Loop Cache Example. IEEE Computer Architecture Letters, Vol 1, Jan. 2002.

[19] Jouppi, N.P., Cache Write Policies and Performance. Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on Computer Architecture, May 1993. pp 191 -201.

[20] Chiueh, T., Pradhan, P., Cache Memory Design for Network Processors. Proc. of IEEE 6th HPCA, Jan 2000. pp 409-418.

[21] Vahid, F., Lysecky, R., Zhang, C., Stitt, G., Highly Configurable Platforms for Embedded Computing Systems. Microelectronics Journal, Elsevier Publishers, Volume 34, Issue 11, Nov. 2003, pp 1025-1029.

[22] Benini, L., et al. From Architecture to Layout: Partitioned Memory Synthesis for Embedded Systems-on-Chip. Proc. of 19th DAC June 2002. pp 777-782.