CACHE MISS REDUCTION TECHNIQUES FOR EMBEDDED CPU INSTRUCTION CACHES

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

By
Kenneth W. Batcher

May 2008
# Table of Contents

List of Figures ................................................................. iv
List of Tables ............................................................................ v
Acknowledgements .............................................................. vi

1. Introduction ................................................................................. 1
   1.1 CPU Cores and the Embedded Environment ......................... 5

2. Cluster Miss Prediction (CMP) .................................................... 14
   2.1 Cluster Miss Prediction Prefetching .......................................... 27
   2.2 CMP Prefetch Architecture .................................................. 28
   2.3 CMP Cache Miss Servicing .................................................. 30
   2.4 Classification of CMP Cluster Misses ...................................... 34
   2.5 Results ................................................................................ 38

3. Prefetch on Miss Combined with Cluster Miss Prediction ................. 41
   3.1 Cache Miss Servicing .......................................................... 44
   3.2 Cluster Profiling ................................................................. 47
   3.3 Experimentation ................................................................. 51
   3.4 Cluster Evaluation ............................................................. 58

4. Interrupt Triggered Instruction Prefetch ......................................... 63
   4.1 Background and Related Work .............................................. 66
   4.2 Interrupt Triggered Software Prefetching ................................. 70
     4.2.1 Interrupt Servicing ......................................................... 71
     4.2.2 Implementation of ITSP .................................................. 73
   4.3 Goals and Motivations ......................................................... 76
   4.4 Experimental Evaluation ..................................................... 78
     4.4.1 PPT Configuration ........................................................ 81
   4.5 Experimental Results ......................................................... 82
   4.6 Conclusion ......................................................................... 84

5. Dynamic Round Robin scheduling ............................................... 89
   5.1 Background ......................................................................... 91
   5.2 Techniques and Implementation ............................................ 93
   5.3 Implementation ................................................................. 98
   5.4 Algorithms Evaluation ....................................................... 101
   5.5 Experimental Evaluation .................................................... 103
   5.6 Summary ........................................................................... 111

6. Conclusion ............................................................................ 112
   6.1 Contribution ....................................................................... 113
   6.2 Future Work ....................................................................... 116

References ............................................................................. 121
List of Figures

Fig. 1. Embedded system with prefetch logic external to the core ............................................................... 9
Fig. 2. Typical embedded system with internal prefetch logic ................................................................. 18
Fig. 3. Flow graph showing a cluster sequence .................................................................................. 21
Fig. 4. Cluster miss identification, cache miss density .............................................................................. 26
Fig. 5. CMP prefetch architecture .................................................................................................... 29
Fig. 6. Cluster miss predication history table format .............................................................................. 31
Fig. 7. CPU activity using cluster miss servicing .................................................................................. 33
Fig. 8. CMP + POM architecture ....................................................................................................... 41
Fig. 9. Various prefetching scenarios illustrating WCET ........................................................................ 47
Fig. 10. Cluster selection algorithm ..................................................................................................... 51
Fig. 11. CMP+POM evaluation method .............................................................................................. 55
Fig. 12. Cluster characteristics for difference sizes and configurations .................................................. 62
Fig. 13. CPU servicing different interrupts .......................................................................................... 65
Fig. 14. Typical ISR with prefetching instructions added ....................................................................... 72
Fig. 15. Jump table for PPT access ...................................................................................................... 73
Fig. 16. Cache misses seen during simulation ........................................................................................ 76
Fig. 17. Reduction in cache misses with prefetching using 16KB caches ................................................ 86
Fig. 18. Reduction in cache misses with prefetching using 4KB caches ................................................ 86
Fig. 19. Two example round-robin schedules for 6 tasks ........................................................................ 94
Fig. 20. Running and tuning states ...................................................................................................... 95
Fig. 21. Typical tuning using dual thresholds ..................................................................................... 99
Fig. 22. Running and tuning states ..................................................................................................... 101
Fig. 23. Instruction cache misses for various positional schedules ....................................................... 105
Fig. 24. DMA based interrupt triggered hardware prefetching ........................................................... 118
List of Tables

Table 1. Profile of four cluster misses in trial application. .................................................................35
Table 2. Sample code profile in trial application. ...............................................................................39
Table 3. Results of CMP simulations. ................................................................................................39
Table 4. Results using different prefetch methods and cache configurations. .................................57
Table 5. Hit rates experienced during simulations. ...........................................................................57
Table 6. Prefetches experienced during simulations .........................................................................61
Table 7. Summary of cluster prefetch performance. ..........................................................................61
Table 8. Prefetch prediction table .....................................................................................................72
Table 9. Test simulation software for Interrupt triggered prefetching. ............................................79
Table 10. Foreground task interference metric comparison. ...............................................................79
Table 11. Overall result of tasks using interrupt triggered software prefetching .............................87
Table 12. Test simulation software for DRRS. ..................................................................................104
Table 13. Positional simulation result summary. ..............................................................................108
Table 14. Prefetches experienced during simulations 16KB caches ...............................................109
Table 15. Prefetches experienced during simulations, 4KB caches ..................................................110
Acknowledgements

This dissertation embodies the conclusion of 9 years of studies and research devoted to embedded systems. I have pursued my studies part-time while working a demanding job full-time, so I have learned how to multi-task very well. I could not have finished this work without the help of my advisor Robert Walker and the support of my company Cisco Systems which funded my studies. My children and family and friends have also been very supportive and I thank them for their support. I wish to thank the University and all the professors I have taken classes from, and especially those professors which have reviewed my work. Overall it has been a lot of fun and worth the effort.
Chapter 1. Introduction

This research began by observing the run-time behavior of industrial wireless applications using embedded microprocessor (CPU) cores. It was discovered that the performance of such systems was less than optimal and often hampered by the restrictive nature of the embedded environment. Further, instruction cache performance proved to be a significant bottleneck in the operation of the microprocessor. Unfortunately, the classic techniques implemented in these cores by the manufacturer or implemented in software to reduce cache misses does not work well for these embedded applications. In the context switching environment, some novel techniques have been developed aimed at improving the cache performance of microprocessors [8][10]. However, the embedded designer can not implement these techniques using licensed CPU cores, since those can not be internally modified. The motivation for this research, therefore, is to discover a practical way to reduce the instruction cache miss bottleneck which works within the confines and constraints of embedded systems.

Embedded microprocessors have enabled a wide range of new products in today’s society. Portable consumer electronic devices such as cell phones, laptop computers, digital cameras, personal digital assistants (PDA’s), and video recorders have become increasing popular and affordable. Embedded systems are also found as digital sub-components in large complex systems such as automobiles, televisions, set-top boxes, IP telephony, and peripherals for computers. These systems are typically low cost, low
power and contain ever increasing functionality. These trends drive the need for embedded system development in order to implement the demands of the consumer.

The term “embedded system” has numerous meanings in the industry. Panda et al [68] provide a good definition of an embedded system, also referred to as “application specific systems” which they define as computing devices created to perform a single or dedicated application. This is opposed to general purpose computing devices such as workstations or personal computers which can be used for a wide range of applications depending on the software running on those systems. Instead, embedded systems contain a small microprocessor with a small memory footprint. Some embedded systems include an operating system, but such operating systems are so small and specialized that the entire function can be written as a single program. The software running on embedded systems is fixed in nature and never changes (except for rare maintenance updates) during the life of the application. Hence opportunities are present to optimize an embedded system by tuning it to efficiently run the fixed software application.

An embedded system is commonly implemented as a System on a Chip (SOC) integrated circuit using standard cell CMOS technology. Integration is done on this scale to meet cost and power objectives. Often referred to as ASICs or Application Specific Integrated Circuits, these custom devices utilize small CPU cores with a modest amount of memory and small on-chip level 1 caches. These CPUs are then connected to an external memory interface which is typically a low cost synchronous dynamic memory or SDRAM.
Other ASIC modules are interfaced with the memory and CPU using an on-chip bus such as the popular AMBA standard [4]. Direct Memory Access or DMA transfers are often used to assist and offload the CPU data movement and are done using custom ASIC modules. The CPU core is licensed and fixed processor logic. Outside this core, the engineer is free to add custom logic. ASIC technology has progressed well through the last 10-20 years to meet the growing demands of embedded systems. Today’s standard deep sub-Micron CMOS is the mainstream technology enabling very complex ASICs to be developed for extensible and flexible embedded systems. These device geometries are so dense that complex embedded systems can even be implemented as FPGAs (Field Programmable Gate Arrays) which can be modified after a product is fabricated.

To summarize, engineers designing embedded systems face a challenging problem that is constrained in the following ways:

1. Low cost, including package cost
2. Real time processing constraints
3. Low power, often power over Ethernet or battery
4. Increasing performance and functionality
5. Small area and I/O pin count

To build a competitive product, engineers must maximize features and performance in a low-cost ASIC-based system. This goal forces them to pack as much functionality as possible onto each chip, increase processor frequencies and push bus bandwidth to the limit. Since schedules are often very tight to gain the necessary time-to-market advantage, the designers often license CPU soft cores to use in their designs.
These flexible and reusable CPU cores, offered by companies such as MIPS™ and ARM™ [88][3], can be configured with internal instruction and data caches and/or scratch pad memories. These cores are termed “soft”, meaning they are implemented using a hardware Register Transfer Language (RTL), so that they can be re-compiled by the user to target different technologies and/or different CPU configurations using ASIC digital synthesis. This is also referred to as *paramaterizable* code, since the synthesis of these devices can be driven from customer defined values in order to tailor the system to a specific need. A big advantage of embedded CPU cores is design reuse, where a mature existing tool base (compilers, assemblers, etc) can be leveraged to speed product development.

With the competitive and fast paced embedded products industry, time to market is very important. If products such as consumer electronics are not created in a timely fashion, the opportunity to make a sale will be lost. By using an embedded processor, one enables a software based product to be developed. This allows for much needed flexibility as compared to developing a purely hardware based device. Thus not only can such products be upgraded in the field to fix issues, but it is also possible to create slightly different devices with a change in the software program. Therefore a different market segment can addressed with a software change. This is a much more cost affective alternative compared to creating pure hardware ASICs. Furthermore a software based CPU device can adapt to the rapidly changing industry standards without the need for expensive hardware based changes to be instrumented.
1.1 CPU Cores and the Embedded Environment

While these CPU cores can be customized by the designer to some degree, opportunities are limited. For example, designers can choose the core’s cache memory size, but must often select modest memory sizes due to the cost and power constraints of embedded systems. Thus besides fixing the physical parameters of the caches (size, ways, etc.), there is little more a designer can do to improve cache performance since the caches are internal to the CPU cores.

Instruction and data caching techniques have been around for over 50 years in the computer industry, especially for general purpose computers [38]. The idea is to use keep current data and frequently used instructions in a small local memory for speedy access. Caches are dynamically managed, meaning the cache contents are filled on demand by memory accessed by the CPU as it executes. A cache will typically dynamically mirror a much larger off chip memory space. Since the cache is small, it often runs with single cycle access at the speed of the microprocessor. Latencies for off-chip memory access are typically on the order of 20-70 cycles, or even worse in some cases for embedded systems, a trend which continues to worsen as memory speeds grow more slowly compared to processor speeds [39] [82]. The big advantages of cache are the performance boost experienced over the entire execution and the transparency of operation to the software. Since the hardware operates behind the scenes, software does not have to be specially written or recompiled to utilize cache. This is why caches are so popular in the industry and used by virtually all types of microprocessors.
Unfortunately, the software instruction space for embedded systems is constantly growing to fuel increasing functionality demands, adding to the need for efficient caching. This trend is fueled by the need for increasing features being consolidated onto a single embedded device. This creates a “Swiss Army Knife” affect as a single embedded device takes on various functions once done by multiple devices; network connectivity is a good example of the prevalent trend. Personal Digital Assistants (PDAs) now come standard with embedded wired and wireless network capability. Cell phones are now being developed to include wireless network connectivity in the form of IEEE 802.11 wireless LAN in addition to cellular network [20].

Furthermore, companies are beginning to offer cell phone functions embedded with PDAs and MP3 players for music. Requirements like these, coupled with the embedded nature of devices create unique challenges for today’s engineers which increase the complexity and size of the software running in the system. Thus embedded applications are constantly growing to support new emerging features with much of the new functionality being added to the pre-existing legacy code to ensure backward compatibility with existing features. This addition “software code bloat” results in an even larger memory footprint, making it increasingly less likely that needed code and data will fit in the small embedded cache.

With embedded systems, a cache miss penalty is especially detrimental since shared busses of narrow width typically serve as the main path to external memory. This memory path is also commonly shared with other ASIC modules that use DMA access to
memory, so non-deterministic arbitration with the CPU cache miss servicing occurs. This sharing adds to the miss penalty and makes the cache and thus performance even more unpredictable.

For embedded systems, real time processing constraints must often be considered. For example, in the networking domain quick reaction to interrupts and predictable packet processing throughput is often more important than raw CPU speed. These constraints make it necessary to bound the Worst Case Execution Time (WCET) to ensure that the system can keep up with demanding packet flows [74]. Real time processing deadlines are often present in these embedded systems, and further, such environments are usually interrupt rich resulting in frequent task pre-emption during execution. Hence context switching is common since the software running on these systems is often multi-tasking nature. Typically a primitive custom operating system is present due to low cost and limited computer memory space available.

Most embedded systems typically run dedicated applications, but even so the data processed by such systems varies widely depending on the environment and system state, so the instruction flow varies erratically. Unfortunately, a dominant factor that affects WCET is the cache performance of the microprocessor [90], so this erratic instruction flow is quite harmful to the performance of the system. Therefore real time processing constraints must maintained in the design of the software and memory sub-system. Many researchers have explored the important issue of quantifying and bounding the worst case
execution time with relation to embedded systems and the software running on them [56][58] [43].

Often embedded applications also have real time processing constraints. Safety critical systems such as industrial automation or defense related applications are another example. In those systems, firm deadlines must be met in order to ensure proper operation. Predictable CPU performance is also required for any embedded system controlling a mechanical device, such as a motor vehicle.

Besides WCET, another important consideration is the overall CPU efficiency while running the embedded application. It is always desirable to have the CPU do as much useful work as possible by reducing the amount of stalling that occurs. When a cache miss happens, often the CPU stalls waiting for the data to be fetched from main memory. This results in slower performance and additional bus activity causing more power consumption. By reducing the amount of stalls, the CPU can gain additional cycles for background tasks and other functions that it would not have time to perform otherwise. Reducing the amount overall amount of stalls can also be thought of as minimizing the average execution time (ACET) for the application as opposed to minimizing the worst case execution time. An ideal memory subsystem for an embedded system will do a good job at optimizing both of these criteria.

This dissertation topic focuses on different types of embedded systems, some with real-time constraints. These applications are 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 input data behavior at a particular instance in time. Features such as interrupt processing and context switching make traditional cache architectures less effective for embedded applications. Typically these applications are also multi-tasking as they must juggle a number of operations to perform their function.

Considering all these factors together, while caches have their advantages, in real-time embedded applications they also have strong disadvantages. The small cache size necessitated by cost constraints leads to high miss rates, and the miss penalty is large due to the shared narrow busses. This large miss penalty causes frequent stalling by the CPU which can lead to missed processing deadlines if the time-critical code is stalled too long. This stalling is especially prevalent when new data arrives in the system (such as a network packet or data sample). This data often invokes a new task or interrupt service routine, forcing the cache to change radically for the new working set. These behaviors degrade the WCET, make cache performance unpredictable, and contribute to increased bus bandwidth – all factors that make caches less desirable in embedded systems. Thus these embedded systems cannot capitalize on all the advantages that cache has to offer.

![Fig. 1. Embedded system with prefetch logic external to the core.](image)
One approach commonly used to reduce the miss penalty and make the cache more attractive is prefetching. This was explored heavily in my research. In an embedded system with soft CPU cores, prefetch logic can be added outside the CPU as shown in Fig. 1. Note the prefetch logic outside the core. This logic assists the hardware that services the cache misses (which is inside the licensed CPU core and cannot be modified) by attempting to anticipate the upcoming cache misses and read the necessary data ahead of time into a buffer. There are many types of prefetching, but most do not work well for embedded systems that run real-time applications with CPU cores.

Cluster Miss Prediction (CMP) is an experimental prefetching method which addresses the unique requirements of these embedded applications. I developed this new concept (Chapters 2 and 3) which profiles the software running in the embedded system in terms of WCET and overall cache miss penalty. Using this information, a cluster can be identified, which is as a string of instruction cache misses that occur in a relatively short period of time and are repetitive throughout the execution. It is these clusters, or bursts of cache misses, that most adversely affect WCET and overall system performance in terms of CPU efficiency. CMP uses this profiling information to load a history table, which helps identify the onset of a cluster and initiate prefetches of cache lines in the cluster, thereby improving cache miss latency. Further advantages of this method will be illustrated and described later in Chapters 2 and 3.

CPU cores often contain built-in prefetch logic internal to the core that can be invoked with special software instructions. A new method more suited to real-time
systems was developed as part of this research that requires no special hardware. In embedded systems, handling time-critical real-time tasks is a challenge. The software may not only multi-task to improve response time, but also supports events and interrupts, forcing the system to balance multiple priorities. Further, pre-emptive task switching hampers efficient interrupt processing, leading to instruction cache misses. My research also provides a methodology called Interrupt Triggered Software Prefetching (ITSP) for using software prefetch instructions in the interrupt handler to improve efficiency, thus making instruction caches more attractive in a real-time environment. The benefits of this technique are illustrated on an embedded ARM processor model running application benchmarks with different cache configurations and interrupt arrival pattern in Chapter 4.

Another software concept was developed in this research which helps reduce cache misses due to conflicts between different tasks. This software technique dynamically adjusts the order of the tasks being executed in a time sliced round robin multi-tasking environment. Through a series of step-wise refinements the software system eventually learns a better temporal sequence for the tasks, one which reduces conflicts between the set of currently running tasks. This low overhead technique called Dynamic Round Robin Scheduling or (DRRS) is demonstrated in Chapter 5 using an ARM processor model which runs application benchmarks and different cache configurations.
The overall focus of my dissertation is *cache miss reduction techniques for embedded CPU instruction caches*. Only embedded CPU systems running dedicated applications are considered, though those are used for a broad range of applications. These research methods are specifically aimed at instruction caches, although some observations were made related to data cache performance during experimentation. Different methods were developed, each with its own unique advantages. Experiments were done to compare and contrast the results among different methods. Since a very similar set of application benchmarks was used for all experiments, this provided a common framework for experimentation. Many different cache configurations and sizes were utilized as well as two different industrial processor cores (MIPs and then ARM) In summary, this methodology illustrates the diversity in the application of this research on realistic designs.

This format of this dissertation is as follows. Chapter 2 describes research related to Cluster Miss Prediction (CMP) techniques. This hardware prefetching method introduces the concept of *clusters* and how they are identified, including formulas, algorithms, results, and descriptions of metrics developed to classify clusters. Much of the material in Chapter 2 was published at the GLSVLSI conference [11]. This technique was later expanded upon with additional hardware to perform Prefetch on Miss (POM), as introduced at the CASES conference [12] and is detailed in Chapter 3. Chapter 4 introduces Interrupt Triggered Software Prefetching (ITSP). In contrast to the previous hardware prefetching methods, this software method interrogates the state of pending interrupts and performs software prefetches based on predictions of upcoming interrupt
service. ITSP is suitable for real-time systems or other systems dominated by interrupt service. ITSP was introduced at the RTAS symposium [14]. Finally, Chapter 5 describes Dynamic Round Robin Scheduling (DRRS). This software technique improves a multi-tasking embedded system by rescheduling a set of running tasks to avoid cache misses. This rescheduling is done dynamically during run-time after the embedded software is compiled and linked allowing the system to adapt to a changing environment. Chapter 5 draws most of its’ material from my fourth and most recent publication at the DATE conference [13]. Finally, a conclusion is provided in Chapter 6 followed by references.
Chapter 2. Cluster Miss Prediction (CMP)

CPU cores running embedded applications operate in a challenging environment. Performance can become hampered by cache misses that cause the processor to stall, preventing the CPU from doing useful work. It is expected the software will experience a certain amount of cache misses. However, when flooded by an unexpected series of back to back cache misses, the embedded software can experience erratic performance. Cluster Miss Prediction (CMP) is designed to prefetch in situations where the CPU stalls due to a cluster of cache misses. I first introduced this material at the Great Lakes VLSI Symposium [11].

To understand how prefetching works, it is useful to overview the principles of cache operation. Caches work on the principle of locality of reference. A good definition is provided in Wikipedia [97]:

“In computer science, locality of reference, sometimes also called the principle of locality, is a concept which deals with the process of accessing a single resource multiple times. There are three basic types of locality of reference: temporal, spatial and sequential:

* Temporal locality - the concept that a resource that is referenced at one point in time will be referenced again sometime in the near future.
* Spatial locality - the concept that likelihood of referencing a resource is higher if a resource near it was just referenced.
* Sequential locality - The concept that memory is accessed sequentially.”

Caches are organized with blocks of data organized as cache lines. Normally 4-16 instruction or data words (of 32 bits each) comprise a cache line. When a cache miss
occurs, a line fill happens where the entire line of data containing the missing datum is fetched from the main memory and placed in the cache, quite possibly displacing the data previously there. Cache *set-associativity* refers to sets of lines grouped logically by address to reduce the lookup space. When displacing data in the cache, the missing line could replace any of the lines belonging to that set causing a cache conflict. Hence caches capitalize on the fact that most CPU memory references have locality and thus can improve performance by holding memory locations which have strong locality close to the CPU in a fast local memory.

When the CPU accesses a memory location which is not sitting in the cache, it is known as a *cache miss*. This causes the location to be retrieved from main memory resulting in *cache miss penalty*. There can be a great deal of latency associated with a cache miss, hence even if the cache miss rate is small (say 1% of all access), the penalty can be on the order of many cycles to retrieve the cache data. Thus it makes sense to concentrate on improving the cache performance: which involves *miss rate and miss penalty* to boost the microprocessor performance.

A number of cache enhancing techniques have been investigated for this research. Focus has been on methods that are popular with embedded systems and methods which could be alternatives to using CMP based method. These can be roughly classified into 3 different types:

1.) Hardware Prefetching
2.) Software Prefetching
3.) Scratch pad memory
Each of these methods will be described in turn below as numerous literatures were examined during this research. Since these methods are used so widely in industry, sources included not only those aimed at embedded systems but also various microprocessor journals and proceedings. It should be noted that a great deal of literature exists on data prefetching as opposed to instruction prefetching. Some of these works were consulted as well [60][28][23] but the focus here is on instruction prefetching methods.

*Hardware prefetching* is a popular technique for enhancing cache performance. The idea is to augment the CPU with a hardware predictor of some type to service the cache for upcoming misses. Typically these predictors examine dynamic information by snooping the instruction flow or other state information inside the CPU to figure out what to prefetch. These hardware methods generally use correlated or non-correlated prefetching. A good overview of various hardware techniques is presented by Hsu and Smith [40]. Some correlated techniques include threaded prefetching, Markov prefetching, branch-history guided instruction prefetching. A common theme of these methods is use of dynamically updated history state information.

Threaded prefetching methods maintain a list of pointers associated with each instruction block to guide the prefetch decision. These pointers are updated with history information as the program executes, adapting to dynamic changes in instruction flow. Therefore past instruction access patterns are correlated to derive the best current prefetch decision when currently executing from a given cache block. Other correlated
methods also use history information, notably Markov prefetching [46]. These methods store dynamic instruction sequences after they have been fetched and executed in order to follow the instructions at subsequent times. Markov prefetching can be used to correlate consecutive miss address whereas branch history methods correlate the execution of branch instructions and their outcome. Execution history guided instruction prefetching is yet another method which correlates execution history with cache miss history [102]. Generally correlated prefetch methods execute prefetches several steps ahead to overcome the cache miss latency since the history patterns are learned from prior execution behavior.

Other hardware prefetch methods do not use correlated history information. These methods are generally less complex since there is no state preserved and trigger the prefetch decision based on a current events rather than predicting based past history. Some examples include fetch-directed instruction prefetching which uses branch prediction of the instruction stream by looking several blocks ahead to guess at where the instruction flow will go next [77]. Another example is wrong path prefetching which also attempts to optimize based on the target of branch instructions [71]. A classic technique is next line sequential prefetching. Here multiple sequential cache lines are prefetched based on the current cache line access to try to stay several steps ahead of CPU. This method is called always prefetch, which capitalizes on the fact that normally instruction flow proceeds in a sequential fashion.
In general most hardware prefetch techniques implement prefetching internal the CPU as shown in Fig. 2 and are not intended for embedded systems. The aim is to reduce the \emph{cache miss rate} by prefetching instructions directly into the internal cache, thereby preventing a cache miss altogether. This does not help embedded applications using CPU cores for two reasons. First, the prefetching may cause cache pollution in these applications, displacing useful data with prefetched data \cite{35}. This could harm worst case performance should useful data be replaced. Second, users of these CPU cores do not have clean access to the internal cache logic to implement these design enhancements. As a result, any prefetching techniques employed in these embedded systems must reside entirely outside the CPU core on the bus between the CPU and main external memory. Thus instead of focusing on reducing the \emph{cache miss rate} like most prefetch methods, in this case I focus on reducing the \emph{cache miss penalty} when a miss occurs.

![Typical embedded system with internal prefetch logic.](image-url)
Prefetch On Miss (POM) with stream buffers is a hardware prefetch technique whereby hardware will fetch the next instruction line in sequential order in reaction to a cache miss. This differs from always prefetch, which triggers on current cache hit accesses. This classic technique has been used quite effectively in a number of applications and works as follows[47][48]. If a cache miss occurs on cache line address A, the hardware will service that miss in the normal manner, but will also prefetch the next cache line address (A+1) as soon as possible thereafter, holding it close to the CPU for the upcoming miss. This method performs well when the instruction flow proceeds in sequential order, in which case each cache miss is normally followed by cache misses to sequential line addresses. Using POM, it may be possible to avoid those cache misses if data is prefetched in time and placed directly into the cache. POM can also be used to reduce the miss penalty if access to the internal cache is not available.

POM is a low-cost technique that can be easily added to embedded systems outside the CPU core, requiring only a small buffer and simple logic. To store the prefetched data, a buffer must be added that can hold at least one cache line. Simple logic must also be added to detect the onset of a cache miss and initiate a bus cycle to prefetch the next sequential address after the true miss is serviced. Fortunately, most CPU cores provide external signals for monitoring cache performance [101], so the cache miss can be detected by prefetch logic outside the CPU core. These signals typically define the type of miss (data or instruction cache) and when the CPU is stalling for cache service.
Fetch ahead distance or pre-fetch ahead distance is a term indicating how deep the prefetch buffer is. This means the prefetch logic sits several cachelines ahead of the actual CPU bus requests. By doing this, it may be possible to avoid even more prefetch latency, however predicting incorrectly may result in a larger penalty affecting WCET and resulting in more wasted bus traffic due to the aggressive nature of the prefetching. Most types of prefetching using external buffers use a fetch ahead distance of several cache lines to make it more affective, including POM and always prefetch.

Unfortunately, while POM is an effective technique in many situations, the presence of a high degree of branching, context switching, and interrupt processing all cause non-sequential code behavior that reduce the effectiveness of POM in embedded applications. Even worse, incorrect prefetching wastes valuable bus cycles and thus can further effect execution time under worst case scenarios. Taken together, the reduced effectiveness of POM and the high cost of incorrect prefetching make POM less attractive for real-time processing.

_Cluster Miss Prediction (CMP)_ is a new method recently introduced for prefetching when code execution does not necessarily proceed in a sequential manner during worst case performance situations. CMP attempts to detect the onset of an instruction cluster miss and prepare a buffer for reducing the miss penalty. 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 that are repetitive throughout the execution. In real-time networking applications on embedded systems with small caches, the observations indicate that instruction cache misses tend to repeat in these clusters, making this a potentially useful technique.

For example, when tracing instruction execution for an industrial networking application, the following sequence of four cache line miss addresses (16-byte aligned address) occurred in a short period of time: 0x480, 0x8C0, 0x9C0, 0x1FC0. This sequence caused much CPU stalling and repeated, occurring many times throughout the execution of the embedded application in a bursty fashion. The cluster in this case did not exhibit sequential behavior, so POM would not have been effective, whereas CMP aided the cache as the cluster was detected.

An instruction flow graph illustrating a cluster of four misses is provided in Fig. 3. The onset of the cluster occurrence was caused by a packet arrival, which in turn
invoked an interrupt service routine and forced the program flow to leave the background task (main). Depending on the outcome of the parse, the Crypt routine might also be called which in turn might call the Signature routine. Eventually the interrupt service finishes and the background task is restored.

Other researchers have also explored software based prefetch methods. With software prefetching, the processor is altered to provide special instructions to initiate prefetching. These can be inserted by a compiler or as the software is written by the developer. An example of such a method is cooperative prefetching which uses software instructions to trigger prefetch hardware to obtain instructions well in advance [59].

Compiler methods, including the in-lining of functions at compile time, can also eliminate some control flow dependencies [64]. However, observations of the test application indicate that worst case situations are often caused by interrupt processing and associated context switching, which is unpredictable at run time or compile time. This renders most software techniques less effective in this particular case, since there is no way to predict when an external interrupt will arrive and cause a context switch. Furthermore some CPU cores may not include software instructions to support for prefetch operations. Even if they do, the scattered nature of the cluster miss addresses renders them less affective.

Other interesting software methods include changing the cache replacement policy for better cache performance and predictability [44]. Previous research involved favoring the caching policy by using a context identifier associated with cache regions to
prioritize different tasks [10][8]. Again, these methods cannot be directly employed with a CPU core due to the intrusive nature. Other researches have experimented with partitioning the cache to allow for better predictability [90]. With this approach, the cache policy is changed to favor certain critical routines for a higher probability of inclusion in the cache so they are more likely to be accessible. Luk and Mowry use a sequential prefetching method coupled with software based prefetching for the targets of branch instructions [59]. This represents a joint hardware and software method which contains good advantages for their target systems.

Some embedded systems use scratch pad memories as an alternative means of improving cache performance, sometimes even combined with caching to get the benefits of both [67]. This method uses tightly-coupled on-chip memories to store time-critical pieces of instruction space that need to execute in real time and have predictable behavior. Unfortunately, this method suffers from the drawback that the code image must be modified to use the scratch pad. Although many utilities and techniques exist to assist with the necessary memory transformations [15][26], they are still not as transparent to the software as traditional caches. Furthermore, most scratch pad techniques seem more appropriate for improving data cache performance, rather than assisting with the instruction cache [29][6][94]. Except for holding stack variables, data cache helped little in my networking application, since the software acts on packets that are accessed typically only once as they transit through the system. However, a small data scratch pad was quite effective at storing active packets and greatly reduced any stalls related to data cache miss. The active packet was placed into the same data
scratchpad location each time by hardware DMA so the CPU could process it without cache miss.

A small instruction scratch pad memory comparable in size to the instruction cache was evaluated during my experimentation. However, the use of the instruction scratch pad to be less efficient at storage of clusters since the ones that adversely affect WCET were often spread throughout memory in a non-contiguous fashion (see Fig.3. page 21, which shows the wide span of cache line addresses). While most embedded CPU cores provide external scratch pad interfaces that co-exist with caches, they usually require that the scratch pad be logically represented as contiguous address space. In this case, the execution threads that most profoundly affected WCET were found to be scattered throughout memory and hence could not be contained in a scratch pad memory of modest size.

In summary, most prefetch techniques are not as effective in embedded systems that have real-time processing constraints and react to stochastic execution flow. Methods previously described such as Markov prefetching or threaded prefetching can overcome the adverse affects of not only branch instructions, but context switching. However these require internal access to the CPU in order to implement as well as the potential for a large amount of history information which is costly in embedded systems. Therefore designers of embedded systems using soft CPU cores are limited in the cache enhancement techniques that they can employ. Instead, the designer must focus on assisting the existing caches in the CPU cores as much as possible.
Other methods previously described such as Instruction History Guided prefetching and Markov prefetching, etc. use history information like CMP. These methods differ from CMP in that CMP uses static profiling information gathered from the embedded system running a fixed software program to quantify the worst case conditions. CMP does not use a learning process to update history information dynamically. CMP is simpler and specifically designed for embedded systems to focus on the very worst miss sequences no matter how rarely they occur. CMP, like POM, reacts only to instruction cache misses visible on an external bus. It does not react to cache hit access patterns to make speculations on useful prefetching, as done by other prefetch methods.

Fortunately for the designers of embedded systems, the fixed nature of the dedicated applications that run on those systems gives their designers an advantage over those designing general-purpose systems. The designer need only worry about the performance of the fixed embedded system, rather than worrying about all general applications. In the design of dedicated embedded systems, instruction traces of the application and profiles of the CPU behavior have proven to be useful in a wide variety of techniques. For example, such code profiling is popular in embedded systems using instruction caches [62] which is a similar analysis approach used for CMP.

As with any prediction mechanism done by software or hardware, predictions normally work, but will fail at times. No prediction technique is 100% accurate, as there are always rare situations that can be attributed to the worst case behavior. CMP is
designed to selectively prefetch only in these situations to improve the observed worst case performance. This is accomplished using a small number of clusters placed into a history table. These clusters are similar to traces or threads used by some prefetching techniques, but typically are caused by interrupts or context switching. These occurrences are not easily visible in an instruction flow graph. As will be described later, if CMP incorrectly predicts a cluster onset, it does not harm the overall worst case performance as compared to no prefetching.

A possible application of CMP is to augment an existing prefetching method already present in a CPU core. So if any hardware or software prefetch method miss predicts the instruction flow, CMP could be used to fill in the gap. This was illustrated in my research where POM+CMP resulted in a stronger technique. Thus CMP can be thought of as a fallback mechanism, where the history information is loaded to prefetch in those cases where other prefetch methods fail.

-------------Not a cluster, too spread out-------------

Fig. 4. Cluster miss identification, cache miss density.
2.1 Cluster Miss Prediction Prefetching

As shown previously in Fig. 2, the Cluster Miss Prediction (CMP) buffer is external to the CPU core, sitting on the shared bus that gives the CPU access to the external memory controller. The CMP logic observes instruction cache miss activity. If a miss occurs and is correctly predicted, the CMP 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 CMP 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 Fig. 4, 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. In a sense a cluster represents a high degree of cache miss
density, whereby compared to the entire set of cache misses over time, a high concentration exists in particular areas.

The CMP concept is described in the remainder of this chapter and illustrated using a sample industrial networking application. The hardware consists of shared bus 32-bit embedded system (similar to what is show earlier in Fig. 2) with a MIPS CPU core, using a 2-way set associative 8KB instruction cache with 32 byte line size [88].

### 2.2 CMP Prefetch Architecture

CMP is optimized to improve performance when servicing a burst of instruction cache 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 such as what is found in networking applications. This is further 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. Situations like these one would prefer to avoid if possible.

Some of the cache misses that occur in these situations are similar to compulsory misses. *Compulsory misses* are those performed when a program is first loaded, when a
Fig. 5. CMP prefetch architecture.

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 CMP 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. The CMP design, shown in Fig. 5 meets these objectives.
2.3 CMP Cache Miss Servicing

To identify and service a cluster miss sequence, CMP uses a *Cluster History Table* (see Fig. 6). CMP 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 miss addresses in the cluster. This differs from other methods which store branching information instead of miss addresses.

CMP operates 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. The Cluster History Table is used to supply the address of where to prefetch next. The Active Miss FIFO acts much like a stream buffer but is filled not sequentially, but based on the contents of the Cluster History Table [47].

For the 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 done 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 CMP 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.

Fig. 6. Cluster miss predication history table format.
The Active Miss FIFO holds four 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. For the embedded application, the goal was to stay under a system memory budget of 1K 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 = 512 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 Fig. 7.

As Fig. 7 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 access must be arbitrated with other ASIC modules. Prefetching with CMP 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” prefetch. 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 CMP thus uses a “prefetch ahead” distance of 1-4 cache lines (up to the maximum FIFO depth). The dual ported nature of the active miss FIFO allows misses to stream from memory.

Provided CMP initiates prefetches ahead of the actual CPU misses with the pipelined fashion shown in Fig. 7, CMP logic 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 CMP altogether as shown with the dashed line previously in Fig. 5. CMP is similar to prefetch on miss (POM) techniques with streaming buffers [47] but with the following notable exceptions:
• CMP can prefetch cache lines in any order rather than sequential prefetching (next cache line address).

• CMP 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 CMP does not react to dynamic behavior.

• CMP only prefetches lines corresponding to a cluster sequence, so it could be referred to as “prefetch on worst miss sequence” rather than prefetch on all misses.

2.4 Classification of CMP Cluster Misses

CMP relies on profiling code execution using instruction traces to determine the best cluster misses to retain in the Cluster History Table. In general, CMP 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. Other researchers have also used code profiling with embedded systems using instruction caches, thereby exploiting the fixed nature of the software [95][61].

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
Table 1. Profile of four cluster misses in trial application.

<table>
<thead>
<tr>
<th>#</th>
<th>Avg Exec Cycles Per Miss</th>
<th>Ocurr-ences</th>
<th>Cluster Size</th>
<th>Average CML Per Miss</th>
<th>Norm.Cluster Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.6</td>
<td>20</td>
<td>4</td>
<td>36.2</td>
<td>13.9</td>
</tr>
<tr>
<td>2</td>
<td>4.11</td>
<td>31</td>
<td>4</td>
<td>35</td>
<td>8.5</td>
</tr>
<tr>
<td>3</td>
<td>3.6</td>
<td>19</td>
<td>6</td>
<td>31</td>
<td>8.6</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>30</td>
<td>5</td>
<td>30.4</td>
<td>5.1</td>
</tr>
</tbody>
</table>

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 CMP 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 the cluster of misses did not degrade CPU execution significantly (e.g., the cache misses are spread out. This gives a metric of temporal locality of each cluster versus normal execution. 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 for the 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 four, 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 this 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 the trial 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 two CPU clocks to obtain the first critical word and resume execution. The subsequent miss prefetching may result in longer stalls 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 the test application, since the CML was quite high relative to CPU execution speed.

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

1. Run the target application in a real world environment with instruction trace.
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 averaging to get the N most significant clusters (e.g. top 1% or 2%).
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 clusters.
7. Rerun the application; see how much CMP 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).

\[
\text{Normalized Cluster Ratio} = \frac{\text{Average CML per Miss}}{\text{Ave Execution Cycles per Miss}}
\]

In a sense, this NCR metric shows the relative degree of the cache miss density experienced with each sequence of cache misses.

2.5 Results

An evaluation of the CMP method was performed by running target software on the embedded networking application and evaluating the performance via simulation. Cryptographic suites [74], network software routines, and packet traces taken from Netbench [63] are used to emulate software processing of packets. The performance of the CMP compared to prefetch on miss techniques (POM) was evaluated. A prefetch distance of 1 and 4 (4 is same prefetch distance as CMP) cache lines were evaluated using the POM method. Metrics were calculated to determine the prefetch efficiency of CMP 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 cache misses were serviced from SDRAM (average 34 clock penalty).

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. Prefetch efficiency relates to the accuracy that

Table 2. Sample code profile in trial application.

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

Table 3. Results of CMP simulations.

<table>
<thead>
<tr>
<th>#</th>
<th>Method</th>
<th>Total Cycles to Service Misses</th>
<th>% Improve</th>
<th>Prefetch Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>POM1</td>
<td>346,287</td>
<td>14.4</td>
<td>79%</td>
</tr>
<tr>
<td>2</td>
<td>POM4</td>
<td>343,501</td>
<td>15.1</td>
<td>72%</td>
</tr>
<tr>
<td>3</td>
<td>CMP</td>
<td>365,001</td>
<td>9.78</td>
<td>100%</td>
</tr>
<tr>
<td>4</td>
<td>Baseline</td>
<td>404,600</td>
<td>--</td>
<td>---</td>
</tr>
</tbody>
</table>
each technique correctly identified the prefetch as the next miss [40]. Thus out of the
11,900 misses, POM1 resulted in 2499 wasted cycles and POM4 resulted in 3332 wasted
prefetches. CMP method resulted in only useful prefetches, but still a decent performance
without the wasted bus bandwidth activity.

For CMP mode, sixteen clusters were identified using the methods described in
Section 2.4. The clusters often occurred when the cache was reacting to a change in
execution behavior, such as crossing program boundaries, invoking a distant sub-routine,
or doing handling context switches. Eight of the cluster misses exhibited poor sequential
but high temporal locality. The remaining eight 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.

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 CMP works in absence of sequential
behavior, it is most effective in situations where POM fails and where bus efficiency is a
premium
Chapter 3. Prefetch on Miss Combined with Cluster Miss Prediction

As mentioned in the conclusion of Chapter 2, the classic Prefetch On Miss (POM) technique can be combined with Cluster Miss Prediction (CMP), using POM for sequential execution cases and CMP for non-sequential worst case situations. These two prefetching techniques complement each other nicely, and have the additional advantage that they can be implemented using a similar small amount of hardware prefetch logic. The CMP+POM concept was introduced at CASES conference [12] and is detailed in the remainder of this chapter. Application of this method is illustrated using a sample industrial wireless LAN networking application (details on this application are provided in Section 3.4).

Fig. 8. CMP + POM architecture.
With only a slight modification to the CMP prefetch logic shown earlier in Fig. 5, one can also accommodate POM as shown in Fig. 8. When an instruction miss is detected from the CPU core, a trigger address lookup compares the missed address to the starting address of each identified cluster. If the comparison is successful, the resulting trigger address is then used to access the Trigger Cache Line Buffer and Cluster History Table. The same Cluster History Table is used as described earlier in Fig. 6, page 31. The operation of this table is unchanged compared to the regular CMP method as is the Trigger Cache Line Buffer. Thus with a successful trigger lookup, the method is identical to CMP.

If the instruction miss does not match the trigger lookup, then the miss address passes directly to main memory to service the miss. After the cache line is obtained and returned to the CPU, a prefetch immediately starts for the next sequential address obtained by incrementing the original miss address – a Prefetch on Miss (POM) that occurs on all misses as long as the CPU does not hit a CMP trigger address. This situation illustrates the complementary relationship of the two techniques. Anytime there is no CMP trigger address hit, the combined method reverts to simple POM, so it can also help in situations where instruction flow proceeds in a sequential fashion, rather than as a cluster of misses.

The Active Miss Cache Line Buffer is large enough to hold a single cache line. Hence the size of the buffer was reduced compared to earlier work since it was found during experimentation that in the worst case scenarios, the cluster of misses come so
close together that there was only time to prefetch a single cache line in most cases. This reduces the complexity and overall size of the prefetch hardware. The buffer is dual-ported so that it can service the CPU misses in a critical-word-first fashion before it is fully filled from main memory. Thereafter, the remaining misses stream into the CPU via external memory with little or no stall penalty. This is a common technique used in many cache support architectures.

Ad hoc experimentation with another sample networking application was used to derive the depth of the Cluster History Table and size of the clusters stored. A Cluster History Table holding 16 entries, each representing a cluster of size 4-6, resulted in a modest hardware cost with a good performance improvement. The remaining combinatorial logic cost was quite small (< 1000 gates), as was the memory hardware costs (which totaled less than 1000 bytes, most of which was the Trigger Cache Line Buffer and Cluster History Table). Below is a summary of the hardware cost in memory bytes:

- 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 = 512 bytes
- Active Miss Buffer = 1 entries = 64 bytes

The CMP Cluster History Table and Trigger Cache Line Buffer can be preloaded in advance based on profiles of code executing on the embedded system. This allows CMP to react to the very first occurrence of a cluster miss, avoiding what amounts to a
cold start compulsory cache miss. Prefetching methods reacting to dynamic behavior often require a first miss to happen before they react to the subsequent misses.

Since embedded applications run dedicated programs, this preloading allows the system to be tuned based on how the code behaved during previous executions. Furthermore, since these tables are not hard coded, they can be changed as necessary to reflect environment, application, or software changes. Later on in Fig. 10, is a description of a heuristic algorithm with statistical methods that can be used to identify the worst case clusters to retain in the Cluster History Table.

### 3.1 Cache Miss Servicing

The key to a good understanding of instruction prefetch techniques is to evaluate the different instruction cache miss scenarios. With an embedded CPU core, when an instruction cache miss occurs, the CPU typically stalls until the miss is serviced from the main memory. These simple CPU cores do not contain superscalar pipeline architectures or support enough instruction level parallelism to exploit the advantages of non-blocking instruction caches or hit-under-miss instruction caches, so an instruction miss usually results in an immediate stall.

Best case refill times from external memory are on the order of 15-40 CPU clock cycles in today’s typical embedded systems\(^1\). A large component of this delay is due to the reduced bus speed and external memory speed compared to the CPU core clock rate. Often the memory and bus systems run about ½ the speed of the CPU core frequency.

---

\(^1\) Assume 0.13 micron CMOS standard cell technology, 250MHz core CPU frequency, and 125MHz burst DDR RAM.
But first, to service the miss, the CPU must win arbitration for the external memory interface, adding to more unpredictability. In the sample networking application and in typical embedded designs, CPU cache service is given priority over DMA traffic, but since the target memory (usually a burst SDRAM) is capable of efficient block transfers, those DMA block transfer are not interrupted, meaning the CPU must also wait for in-progress DMAs to complete. This creates even more latency.

An occasional instruction cache miss is acceptable even for embedded real-time applications. However, in scenarios where misses come close together and take a long time to service, a real-time system suffers the most since the stalling seriously affects the performance of the entire system. CMP relies on profiling the Worst Case Cache Performance (WCCP) over the span of a cluster of misses, by logging performance monitoring signals as the application executes. The goal is to identify the worst clusters and store their addresses in the Cluster History Table, so as to improve the WCCP and thereby provide more predictability and better cache performance.

Fig. 9 illustrates three different scenarios where prefetching helps to improve cache refill latency. The top example shows the WCCP during a sequence of four instruction cache misses with no prefetching. When the first miss occurs, the CPU stalls while the cache line is retrieved from main memory (“cachefill”). Subsequent misses result in similar stalls, since the latency to service the cache miss is not hidden. The second example illustrates CMP, where the first miss results in a much smaller stall since the miss is serviced immediately from the Trigger Cache Line Buffer (“CMP hit”). If
CMP correctly predicts the onset of a cluster and prefetches correctly, subsequent cache
misses result in prefetch hits where the miss is serviced from the Active Miss Cache Line
Buffer (“buf hit”), again hiding the latency of external memory access. The third example
illustrates POM, which helps in a way similar to CMP, but where the first miss results in
a long stall since no Cache Line Buffer holds the needed data, although prefetch hits
cause subsequent sequential misses to be quickly returned to the CPU.

In these situations, the performance of the system can be characterized by the
amount of time required to service the cluster and the amount of stalling during that time.
Frequent stalling by the CPU results in increased time to service the cluster and hence
increased WCCP. In the case of POM and CMP, prefetching can help reduce the miss
penalty due to stalling, even if the prefetch does not finish in time. A correct prefetch
enables the bus cycle to initiate early so that the miss data is already on the way before
the CPU actually asks for it; these are referred to as partial prefetch hits.

In most cases, inaccurate prefetches are harmless, except for the wasted bus
bandwidth they cause, but occasionally a bad prefetch will actually hurt WCCP. If a
prefetch is started which is wrong, and the next CPU miss occurs during this prefetch
time, the refill penalty is worse since the wasted prefetch must complete before the
correct one happens. This is a situation that the combined CMP+POM method can help
avoid.
Fig. 9. Various prefetching scenarios illustrating WCET.

The experiments I conducted showed instruction miss clusters with evidence of both sequential and non-sequential behavior. CMP was used for those clusters exhibiting non-sequential behavior. POM was used for sequential code, as well as for sequential clusters since it was sufficient to service those clusters (meaning sequential clusters were not added to the Cluster History Table). Thus POM and CMP jointly reduced the WCCP for this application.

3.2 Cluster Profiling

The CMP+POM method relies on profiling the application code execution with realistic packet mixes to determine the best cluster misses to retain in the Cluster History Table. In general, the CMP looks for worst case scenarios where the instruction cache using the POM method performs poorly executing the embedded application. Since
embedded applications typically run a dedicated application program, preloading the Cluster History Table is an effective technique to maximize performance. This preloading also allows the table to be tuned for different applications and environments.

Misses can be characterized using the following four metrics:

2. Cluster Stall Count: instruction stall cycles experienced during the cluster.
3. Cluster Occurrence Count: number of times the cluster happens in the trace.
4. Non-sequential Behavior: flags cases where POM is less effective.

A good cluster candidate for the Cluster History Table has a small Cluster Cycle Count (misses are close together), high Cluster Stall Count (large penalty for cache misses), and a high Cluster Occurrence Count (frequent repetition of the cluster through the execution). Further, clusters must exhibit non-sequential behavior, since sequential clusters can be adequately serviced using POM.

As the application code executes, a detailed performance monitor logs the Cluster Cycle Count, Cluster Stall Count, and miss address for every cache miss. The monitor also carefully evaluates the external bus traffic to gauge the effect of CPU stalling during misses, full or partial prefetch hits, and DMA stalling. The resulting large log file is parsed and processed by Perl scripts that look for repetitive sequences of cache miss address to identify the presence of clusters. Clusters are then ranked as described below, and the best candidates are preloaded into the Cluster History Table.
A primary measure for ranking the clusters is the **Cluster Stall Ratio (CSR)**, the ratio of the time spent on stalls to the total execution time for the cluster (Cluster Stall Count / Cluster Cycle Count). This is a slightly different and more direct way of ranking clusters compared to NCR introduced previously in section 2. A cluster with a high CSR will spend a disproportionately large amount of time servicing cache misses compared to the average amount of cache misses experienced during the entire run-time. So a high CSR makes it a good candidate for the Cluster History Table and hence holds clusters having a high degree of cache miss density. For example, see Fig. 9, where the percentage of time spent stalling is far greater in the no prefetch cases than the CMP or POM prefetch cases. This metric is similar to that in previous work with CMP described in Section 2, but to include POM effects as well, so one can now compare the penalties for the entire cluster rather than only the execution cycles for each CMP miss. Further, CSR is normalized to fairly compare clusters of different sizes (4, 5 or 6 misses) since both CSR and Cluster Cycle Count scale with regards to the number of misses.

Other factors can also be considered in ranking the clusters. The Cluster Occurrence Count to break ties in those situations where two clusters had the same CSR. Since the primary goal was to improve the WCCP, I did not concentrate further on the Cluster Occurrence Count; since a worst case scenario may occur infrequently but may have great effect on predictability and cache performance. In previous CMP work [11], clusters were selected with the goal of reducing the overall Cluster Stall Count along with WCCP improvement. This reduction was accomplished by ranking the clusters according to the product:
Cluster Stall Count x Cluster Occurrence Count.

The cluster selection algorithm begins with a cluster size of 4 misses and selects a candidate cluster at the top of the cache miss address list. It then walks the entire list, and records the penalty and arrival time whenever that cluster occurs in the trace. It then selects the next group of 4 addresses as the candidate, and scans the list again. After each pass through the list, it retains the top candidates in rank order, and when the loop terminates, a final ranking is generated for that pass. This process repeats for cluster sizes of 5 and 6 as well to avoid missing smaller clusters inside larger ones (for example, a particularly bad cluster of 4 misses might get missed if nested inside a 5 or 6 miss cluster). The algorithm runs in $O(n^2)$ time, where $n$ is the number of cache misses experienced during the run.

After all cluster sizes have been evaluated, the highest-ranked clusters are preloaded into the Cluster History Table. Fig. 10 illustrates the algorithm, assuming $c$ as the cluster size. In this code, miss(), penalty(), and time() are lists of cache line addresses, stall cycles, and cycles between misses, respectively, taken from the performance logging. Section 3.4 describes in more detail the results of cluster selection based on using this algorithm.
Fig. 10. Cluster selection algorithm.

3.3 Experimentation

In addition to using this simple selection algorithm, it is possible to customize the cluster selection process to improve one or more specific pathological worst cases. For example, if some other type of static code profiling locates a very bad known sequence,
this cluster can be hand selected for inclusion into the Cluster History Table in case it should ever occur. None of the clusters were hand selected in experiments; instead all clusters were found using the algorithm shown in Fig. 10.

To evaluate the combined CMP+POM method, a cycle-accurate RTL simulation was performed with the CPU core running an embedded wireless LAN driver application – an application that encrypts and formats frames for transmission, and decrypts and parses received frames. For this simulation, the application processed randomly generated frames, in an appropriate distribution to emulate the packet mix seen in typical wireless LANs environments. This packet mix included management, control and data frames, where many of the data frames were encrypted using different encryption methods. The embedded application software consisted of 34 K byte lines of code.

The embedded system hardware consisted of a 32-bit MIPS 4KE soft CPU core and an external DDR memory controller shared with various DMAs running the AMBA protocol. The MIPS 4KE CPU core supports a configurable cache, with a cache line size of 16 bytes. To evaluate the affect of different instruction cache architectures, four different cache parameters were used to observe the cluster effects resulting from differences in miss behavior. The maximum instruction cache size used was 8K bytes since larger sizes were not practical given the cost constraints of the embedded system. In the best case with no arbitration for the main memory, a cache refill from DDR RAM back to the processor took 14 clock cycles, in worst case more than 30 cycles. A hit in the Active Miss Buffer always provides a predictable three clock cycle latency.
Overall the CPU and memory subsystem is similar to that used as described earlier in Section 1.1, except here different cache configurations are evaluated. The software tested with this newer research is quite different. Instead of suites from Netbench, the software was an application specific software driver. However, the packets processed simulated a packet mix seen by an actual wireless LAN application. By using a realistic data stream, the fixed application software was exercised in the same way it would be used in the real embedded environment.

After profiling the execution and selecting the clusters from the baseline case, the simulation was re-executed with three different combinations of options: CMP, POM, and CMP+POM. To avoid biasing the results based on a fixed packet stream from the baseline run (seed A), a different random seed for the random packet generation was used for the three subsequent simulations: CMP, POM and CMP+POM. Each of these simulations used the same random seed (seed B) to fairly compare how each reacted to the same packet flow. Fig 11 describes the simulation flow that was repeated four times for each instruction cache scenario, a total of 16 simulations.

Table 4 describes the different characteristics for each scenario when running for the same amount of simulation time (0.5 Seconds or 100 Million CPU cycles). The “Stall Cycles” columns represent the cumulative amount of CPU instruction stall cycles due to instruction cache misses experienced during each run. The “Accuracy” column represents the total cumulative percentage of accurate prefetches (correct CMP prediction and/or POM sequential prefetch prediction). The “Worst CSR” column indicates how
often the CPU stalled due to the worst observed cluster of any size in that run. Without any CMP, the Worst CSR approaches 100%, which means at some time during the run the CPU stalled continuously throughout the observed cluster interval.

An accurate prefetch for CMP occurs each time the CMP hardware correctly predicts the next miss. Thus for a cluster size of 5, if only the first 3 are correctly predicted, this accounts for 3 hits and two misses. In this way, partial CMP hits are partially counted as useful prefetches, since they still help performance as described later.

POM’s Accuracy column in Table 4 shows a correlation between the increase of sequential misses and the increase in way size (also called associativity), especially with the 4k caches. For example, with a 4k cache size the 2-way set associative cache had 53% of all misses with sequential behavior, but using a 4-way instruction cache of the same size, 79% of all misses were sequential. This suggests that non-sequential miss behavior associated with clustering can be reduced by increasing the associativity, thereby making POM more effective at reducing overall stall cycles. However, changing the associativity has little effect on CSR, with no improvement in 4k case, and only mild improvement from 99% to 97% with the 8k cache size. I also used *POM Accuracy* as a metric to determine the amount of sequential vs. non-sequential behavior experienced in their individual application. In this way, one could gauge the maximum possible effectiveness of POM and how much CMP might help with reducing overall stalls during non-sequential behavior.
The WCCP, tracked using CSR in Table 4, shows POM getting slightly worse compared to the no prefetch case due the presence of inaccurate prefetches that occur at inopportune times. In contrast, CMP helps improve effectiveness by eliminating the 16 worst case clusters that do poorly with POM-only servicing. However, using only CMP does little to help reduce the overall stall cycles, so the performance improvement of
using POM should not be overlooked. POM works well provided the worst case scenarios are addressed with CMP to smooth out the performance.

Table 5 shows the instruction cache hit rates experienced by the application using different cache configurations and prefetch methods. As expected, a larger instruction cache size has a better hit/miss ratio. Keeping the cache size constant and increasing the cache associativity parameters also improves performance, especially in smaller cache sizes. However, recall that Table 4 shows that varying the instruction cache associativity has little effect on WCCP, represented by the Worst CSR column in that table.

Overall, prefetch accuracy was the highest using the combination of CMP+POM. The CMP method did show evidence of inaccurate prefetches, especially with the 4k 2 way cache. CMP guessed incorrectly in cases where the trigger and often the first 1-3 misses are correctly predicted, but the last clusters miss prefetches are incorrect. The hardware was designed to stop prefetch after the first miss and thus a maximum of one failed prefetch is possible with CMP. Since the trigger miss is always correct (or else it would not be a cluster) and serviced quickly from the Trigger Cache Line Buffer, this speedup compensated for the failing prefetch later on. This works since CMP caches the \textit{data} for the trigger cache line and the \textit{address} for the remaining lines in a cluster. Thus CMP does not appreciably affect performance if it guesses incorrectly and usually helped performance, since the benefits of quickly returning the trigger cache line \textit{data} and often starting cluster misses could be still be realized.
Some experimentation was done to explore prefetch ahead distance making the Active Miss Buffer a larger FIFO with more cache line entries. However, it was found that this did little to help, since the CPU demands the next miss too soon in the worst case scenarios. Thus in many of these cases, only one instruction cache line could be prefetched or partial prefetched anyhow before the next miss happens.

Table 4. Results using different prefetch methods and cache configurations.

<table>
<thead>
<tr>
<th>Cache Org.</th>
<th>No Prefetching</th>
<th>CMP</th>
<th>POM</th>
<th>CMP+POM</th>
</tr>
</thead>
<tbody>
<tr>
<td>4k (2 way)</td>
<td>99% 8.62M 46% 89% 7.38M 53% 99% 7.28M 61% 89% 6.11M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4k (4 way)</td>
<td>99% 4.30M 83% 85% 4.24M 79% 99% 3.10M 81% 85% 3.09M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8k (2 way)</td>
<td>98% 1.83M 76% 79% 1.78M 77% 99% 1.29M 82% 79% 1.24M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8k (4 way)</td>
<td>95% 1.77M 82% 76% 1.73M 81% 97% 1.26M 83% 76% 1.22M</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5. Hit rates experienced during simulations.

<table>
<thead>
<tr>
<th>Cache Config</th>
<th>Hit Rate Through CMP</th>
<th>Hit Rate Through POM</th>
<th>Hit Rate Through CMP+POM</th>
</tr>
</thead>
<tbody>
<tr>
<td>4k (2 way)</td>
<td>94.75% 96.5% 97.5%</td>
<td>98.0%</td>
<td></td>
</tr>
<tr>
<td>4k (4 way)</td>
<td>97.2% 98.0% 99.4%</td>
<td>99.5%</td>
<td></td>
</tr>
<tr>
<td>8k (2 way)</td>
<td>97.4% 98.5% 99.4%</td>
<td>99.5%</td>
<td></td>
</tr>
<tr>
<td>8k (4 way)</td>
<td>98.9% 99.2% 99.7%</td>
<td>99.8%</td>
<td></td>
</tr>
</tbody>
</table>
3.4 Cluster Evaluation

For all simulations, evidence of clustering does indeed occur. Clusters were evaluated based on two criteria: improvement in overall miss penalty and improvement in WCCP (improved CSR) using the algorithm described in Fig. 10. What was evident is that regardless of cache configuration and cluster size (4 to 6 misses), a few clusters (<10) contributed the most to the relative CSR as well as overall miss penalty, so by targeting only these with CMP, a decent performance boost is experienced. Overall the CMP+POM method results in good performance improvement compared to the baseline case of no prefetching for all cache configurations. Fig. 12 describes some of these interesting characteristics with regard to clusters.

Each graph shows different configurations where the top 30 clusters are represented. The amount each cluster contributes to improvement in stall time (penalty x occurrences) or CSR is indicated on the Y axis. Larger instruction cache configurations show a steeper drop in CSR and Penalty as the top clusters are removed, but the same pattern exists.

Different clusters with different triggers occurred for different cache parameters despite each simulation having the same software and packet mixes. This makes sense since the cache behavior is strongly dependent on how the misses occur. For example, a miss with one cache configuration may not happen at all if a larger cache is used. These evaluations lead to the justification of the cluster size 4-6 and size of the history table. Thus by targeting the first 16 clusters of size 4-6 a good performance boost is provided at a reasonable hardware cost.
Table 6 quantifies the total number of instruction prefetches experienced during the simulations with the different cache configurations. Since POM occurs on every miss, the POM column also indicates the total number of cache misses experienced. The prefetch quantity is further broken down by the amount of useful prefetches that occur. CMP accounts for far less prefetching compared to the more aggressive POM approach. Since only sixteen clusters are used, only 16 different miss addresses cause the potential for CMP prefetching. However with only a small amount of additional prefetching, a much improved performance in terms of CSR is observed compared to POM alone.

As previously stated, clusters were chosen to address both improved CSR and improved overall stall behavior. CSR was the first selection criteria used until the improvement of the CSR started to plateau. Beyond that point, the incremental affect of adding additional clusters did not appreciable affect the overall worst case performance. Any remaining clusters were then chosen to then address overall cache miss penalty up until the maximum cluster count, in this case it was 16 clusters. For the 4K cache sizes, all clusters were chosen to optimize CSR as no plateau was every reached. But for the 8K cache size, one could attain a good improvement in CSR (from 99% to 80%) with between 10 and 12 clusters as shown in Fig. 12. In general, the cluster selection criteria can be customized depending on the needs of the application, using the algorithm in Fig. 10 as a guide. It makes sense to select the clusters which create the most improvement as determined by simulation.
Table 7 summarizes 16 clusters selected for the experimentation. The most common cluster size was four instruction cache misses as they attributed to the most performance improvement resulting from use of the cluster selection algorithm. Clusters of size 5 or 6 were rare as they would often contain a cluster of size 4 inside. In some cases the additional cache misses to extend a cluster of size four would dilute the magnitude of the cluster, as the additional misses appeared to be more spread out in time. Hence a cluster of size 4 could get missed by adding this additional miss, causing it to no longer be one of the worst case items. This suggests focusing on only clusters of size 4 may be sufficient.
Table 6.  Prefetches experienced during simulations

<table>
<thead>
<tr>
<th>Cache Config</th>
<th>POM (useful)</th>
<th>CMP (useful)</th>
<th>Combined (useful)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4k (2 way)</td>
<td>422,060 (223,695)</td>
<td>75,960 (34,942)</td>
<td>498,030 (258,638)</td>
</tr>
<tr>
<td>4k (4 way)</td>
<td>214,025 (169,080)</td>
<td>4,375 (3,631)</td>
<td>218,400 (172,711)</td>
</tr>
<tr>
<td>8k (2 way)</td>
<td>89,400 (68,850)</td>
<td>3,706 (2,817)</td>
<td>93,122 (71,667)</td>
</tr>
<tr>
<td>8k (4 way)</td>
<td>72,080 (58,385)</td>
<td>2,988 (2,450)</td>
<td>75,070 (60,835)</td>
</tr>
</tbody>
</table>

Table 7.  Summary of cluster prefetch performance.

<table>
<thead>
<tr>
<th>Cache Config</th>
<th>CSR Select</th>
<th>Overall Penalty Select</th>
<th>Size 4</th>
<th>Size 5</th>
<th>Size 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4k (2 way)</td>
<td>16</td>
<td>0</td>
<td>13</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4k (4 way)</td>
<td>16</td>
<td>0</td>
<td>12</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>8k (2 way)</td>
<td>12</td>
<td>4</td>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8k (4 way)</td>
<td>10</td>
<td>6</td>
<td>15</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Fig. 12. Cluster characteristics for difference sizes and configurations.
Chapter 4. Interrupt Triggered Instruction Prefetch

When experimenting with CMP techniques as described in Chapters 2 and 3, I noticed interrupts were a chief source of disruption for the cache operation causing the clustering effect of cache misses. This makes sense due to the task pre-emption required to invoke the interrupt service routine and return to the previous operation. Hence the motivation for Interrupt Triggered Instruction Prefetching (ITSP) which I introduced at RTAS [14] was to develop a practical means of handling this dilemma in an interrupt rich, real time environment. As the complexity of embedded systems has increased, it is now common for those systems to run large software programs that combine such functions as network, voice, and video applications in a single software system. Further, these systems often contain a mix of real-time and non-real-time background tasks [2]. In such soft real-time systems, lives are not at stake if the system fails to meet its deadlines, but failure can result in decreased performance and poor quality of service. In addition system crashes are possible, resulting in unhappy end-users and busy technical support personnel.

Both the hardware and software influence the effectiveness of these embedded systems with regard to how well they operate in a real-time environment. On the hardware side, CPU core vendors have begun to address real-time embedded systems by providing new hardware features for their processors, such as scratch pad memories, multiple banks of registers for efficient context switching, and fast interrupt response
architectures. On the software side, lightweight real-time operating systems (RTOSs) provide an environment that handles most of the complexity associated with a multitasking kernel. For example, a lightweight RTOS provides thread support, time slices, critical section protection, interrupt disable and enable, context save and restore, and various compiler optimizations [53].

One method for improving performance in embedded systems with real-time constraints is to reduce the Worst Case Execution Time (WCET) — the longest path through a code segment [76]. Minimizing the WCET of an interrupt service task is particularly important, since interrupts are typically disabled or queued during this service time, preventing other interrupts or background tasks from executing [19]. Hence the goal here is to execute the interrupt handler quickly, thereby minimizing the amount of time interrupts are disabled (between interrupt service and exit).

It is no surprise that a sequence of instruction cache misses can degrade the WCET [65] [7]. This degradation often occurs when an interrupt-driven context switch pre-empts the current task and disrupts the instruction cache. This is known as inter-task interference, and it results in conflict and capacity cache misses [42]. Unfortunately, the cache miss penalty in an embedded system is often particularly severe due to restrictions on the external memory subsystem servicing the cache. Taken together, these behaviors make cache performance unpredictable and hence less desirable for real-time processing.
Due to this unpredictability, some systems avoid caches and seek other solutions to better accommodate real-time constraints. Scratch pad memories can help, but have limited size and often cannot hold the entire software image. Further, the instruction flow patterns in the worst case path are often not contiguous due to branches and interrupts, which makes methods relying on fixed address space less attractive.

By not using a cache, these systems lose the performance gains and dynamic memory management typically attributed to caches. As a result, the processor cannot be used to its full potential and the overall embedded system performs less useful work. As software size and complexity expands to meet the growing functionality requirements of embedded devices, using a cache becomes more and more desirable [45].

This paper introduces the use of interrupt-based software prefetching for instruction caches as a means to improve performance in a real-time embedded system. Execution profiling is used to characterize patterns of instruction cache misses caused by
interrupt-driven context switches. Thus the system’s prefetching is interrupt triggered, with the pending interrupt information used to guide software prefetch decisions.

4.1 Background and Related Work

Interrupts are a necessary component in many embedded systems that handle real-time constraints. The interrupt serves as the binding between the hardware and software worlds, as real-time hardware events request the services of software running on the CPU. Such events may include timers, exception conditions, interface data requests, and DMA transfer completion. The resulting interrupts have diverse characteristics and priorities that must be carefully handled by both the operating system and interrupt service routines.

Most embedded CPU cores and associated operating systems are multi-tasking, but only service one interrupt at a time. During an interrupt service routine, interrupts are disabled and the operating system usually uses the run-to-completion model where the interrupt service routine does not relinquish control of the CPU until it finishes. Fig. 13 illustrates how a typical microprocessor leaves a background task to service three arriving interrupts.

Embedded systems may also support multiple kinds of interrupts [17]. In some cases the interrupts have a predictable arrival pattern, such as those from a periodic timer. In other cases aperiodic or asynchronous interrupts appear with no warning or regularity, such as those from a network interface. Typical embedded systems support both kinds of
interrupts, which overlap at times and result in multiple pending interrupts waiting for CPU action.

In these interrupt-rich environments, it is common for real-time embedded systems to have a mixture of background task and interrupt service tasks with varying priorities competing for CPU time. A delicate balance must be achieved to prevent starvation yet provide CPU time for all tasks [83][57].

Real-time system designers are also concerned with the amount of time interrupts are disabled. A long service time for a particular task can cause higher latency for other tasks. Fig. 13 for example shows pending interrupts since they cannot be serviced until the current task is complete, for example, note the excessive time between INTA being asserted and being serviced due to the interrupt latency incurred while INTC and INTB are demanding CPU attention.

Further, it is possible that switching between these various tasks due to interrupts can subject the CPU to what is has recently been called interrupt overload [75]. In these extreme cases, the context switching overhead is so severe that the CPU spends most of its time just switching between interrupts. One solution for reducing the overhead in the context switching process is to reduce the number of stalls created by instruction cache misses.

To improve the performance of time critical routines, some embedded systems use managed caches or scratch pad memories [26][6][90]. Some research also involves static or dynamic locking of cache lines [18][5]. Other systems use prefetching to
improve instruction cache performance, and there is a large body of research devoted to both hardware and software prefetching techniques [71][51]. The idea behind prefetching is that the system should fetch a cache line before it is actually needed, and thus prevent a cache miss.

Today’s CPU cores often provide non-blocking software prefetch instructions to force a cache fill at a specified address in anticipation of an upcoming cache miss [69]. However, using these prefetch instructions can be problematic — what are the best locations to prefetch, and how should prefetch instructions be inserted into the software? How much prefetching should be done, and how much will it help? In the context of real-time embedded systems, the designer has little guidance in applying prefetch instructions. Hence the proposed Interrupt Triggered Software Prefetching (ITSP) is a method to help to address the above questions [35].

Observations indicate that with real-time embedded applications, the worst case situations are often affected by interrupt processing and associated context switching, which is unpredictable at run time or compile time. This unpredictability renders many prefetch techniques less effective when there is no way to predict when an external interrupt will arrive and cause a context switch. Therefore real-time embedded systems require special strategies for the use of prefetching.

All prefetching techniques are concerned with prefetch accuracy — the percentage of time when the prefetch actually helps to predict an upcoming cache miss in a timely fashion. If an element is prefetched, but it is already in the cache, is prefetched
too late, or is never actually fetched by the CPU, the prefetch just adds unnecessary overhead. Similarly, the overhead of the prefetch instructions must be balanced again potential savings. Finally, aggressive prefetching can lead to cache pollution and even make performance worse.

In Chapters 2 and 3, _Cluster Miss Prediction_ (CMP) is described as a form of hardware prefetching, designed to improve performance during a cluster of instruction cache misses. 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 execution of the program. The previous observations with embedded real-time networking applications indicate that instruction cache misses are bursty and tend to repeat around context switches.

CMP uses performance code profiling measurement to monitor the instruction cache miss behavior and to detect worst case conditions. A list of worst case clusters is obtained and then used to fill a history table, which guides the hardware prefetching technique. When the start address of a cluster is detected, the hardware will start prefetching the other cache lines of the cluster and thereby reduce the cache miss penalty.

Like CMP, the _Interrupt Triggered Software Prefetching_ (ITSP) method described in this chapter measures performance data gathered during code profiling to identify cache miss sequences. However, the prefetch here is interrupt triggered with the pending interrupt information used to guide the software prefetch decision, rather than using the cache miss address as in CMP. Furthermore the interrupt handler software is
augmented with *software prefetch instructions* to target cache misses, rather than using dedicated *prefetching hardware* as in CMP. This results in a flexible, easy-to-use use technique that can be applied to many embedded systems.

### 4.2 Interrupt Triggered Software Prefetching

No matter how well the software of an embedded system is written or the memory subsystem is designed, some instruction cache misses will occur if the internal memory cannot hold the entire program due to capacity cache misses. Furthermore conflict cache misses between different tasks are unavoidable. However, if the embedded system can be designed to at least reduce the number of cache misses in the presence of context switching, the overall interrupt efficiency of the system will be improved. Other improvements may include reduced interrupt disable time, shorter interrupt latency, quicker service time, and less context switching overhead — all features that are very important to real-time embedded systems.

This chapter describes *Interrupt Triggered Software Prefetching* (ITSP) — a method for real-time embedded systems that adds prefetch instructions in strategic locations of the software to improve the observed worst case performance. This purely software method requires no special hardware, in contrast to the hardware overhead and internal CPU changes that might be required for hardware prefetching. Since such hardware changes are not practical or even possible when using licensed CPU cores, it is more advantageous to use existing features provided by the CPU core wherever possible.
Interrupt Triggered Software Prefetching also allows a pre-existing hardware platform to be improved even after it is released to production, with a simple software upgrade.

Since embedded software does not usually change frequently, code execution profiling is a practical method to measure performance [72]. First, the system is used in the true application environment and its performance is observed. Using this performance data, the interrupt service handler is then tuned using prefetching to target problem spots. The modified software can then be re-executed with the prefetching enabled to improve the performance.

4.2.1 **Interrupt Servicing**

Servicing a hardware interrupt is a complicated process involving the external hardware, CPU, and operating system (OS). When the interrupt arrives, the CPU switches from user mode to kernel/supervisor mode and vectors to an interrupt service routine (ISR). To preserve the state of the previously running task, the ISR pushes the contents of the CPU registers onto a stack (from which they can later be restored when the ISR finishes). The ISR may also need to run various OS maintenance tasks at this time, including thread control, heap management, flushing branch prediction status, and interactions with peripherals. The interrupt is then interrogated to select the appropriate interrupt handler to actually service the interrupt. Finally, once the interrupt handler finishes, cleanup occurs and previous task state is restored.

It is quite possible that interrupts will continue to arrive while the ISR or interrupt handler is running. In the run-to-completion model, interrupts are disabled upon entering
the ISR and re-enabled on exit. In this situation, new interrupts will be queued for service and remain pending until the current interrupt is serviced, at which point the ISR will immediately re-trigger to process the next interrupt in the queue.

**Interrupt Service Routine with Prefetch:**
- read pending interrupt
- access PPT history for current PID
- prefetch based on previous task
- push register data, perform OS maintenance, change PID to new task
- disable interrupts, branch to interrupt handler
- if interrupts are pending, branch back to top of ISR
- restore register data, change PID back to previous task
- re-enable interrupts, return to user mode

Fig. 14. Typical ISR with prefetching instructions added.

### Table 8. Prefetch prediction table

<table>
<thead>
<tr>
<th></th>
<th>Prefetch Target Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0x82C0_4650, 0x82D1_4650, 0x82C0_8800, 0x82C0_4670</td>
</tr>
<tr>
<td>3</td>
<td>0x84C0_9650, 0x82D1_4654, 0x82C0_8800, 0x82C0_4670</td>
</tr>
<tr>
<td>4</td>
<td>0x839C_9660, 0x839C_A660, 0x839C_9686, 0x839C_D660</td>
</tr>
</tbody>
</table>
4.2.2 Implementation of ITSP

ITSP works by tuning the software to be executed based on observed performance during previous executions. There are three steps to this tuning:

1. History collection
2. Cache miss evaluation
3. Code instrumentation

In the history collection step, the system is profiled while it runs the system’s dedicated software in the real application environment. For this step, the following three data items are logged:

- Instruction cache miss address with timestamp
- Interrupt occurrence with timestamp
- Process ID (PID) change with timestamp
This information can be captured in various ways, including logic analyzer traces or cycle accurate simulations as suggested by other researchers [86]. If the CPU core contains internal performance metrics, diagnostic software may be able to capture the necessary information from those metrics. Whichever method is used, the run time should be sufficiently long to obtain an accurate snapshot characterizing the system behavior.

This history data allows a sequence of instruction cache misses to be associated with each interrupt handler, for each previously running process. Offline scripts are used to process this information and detect cache miss sequences that most adversely affect the observed Worst Case Execution Time (WCET). This analysis process is described in more detail later in Section 4.4.1 through a working example.

Using this information, the software in the interrupt service routine (ISR) can be altered with the addition of prefetching instructions, as shown in Fig. 15. The prefetch operations (shown in **bold**) are executed as early as possible in the ISR so that the prefetch instructions dispatched to the cache controller can complete before the interrupt handler is invoked.

The key data structure for Interrupt Triggered Software Prefetching is the *Prefetch Prediction Table* (PPT), illustrated in Table 8 for a sample Task 1. Each task has its own PPT, which contains the prefetch target addresses for every possible task that can run before the current task. For example, if Task 1 is invoked by an interrupt and
pre-empts Task 3, that line of Task 1’s PPT points to the best prefetch targets for the Task-3-before-Task-1 switching combination.

The PPT for each task is configured in the cache miss evaluation step of the software tuning process, with the best prefetch targets chosen according to the information collected in the history collection step. If no cache miss sequence was observed for a particular task switching combination, that PPT entry is NULL. PPT entries may also be NULL if it is impossible for a particular task to pre-empt another, for when priorities and background tasks are considered.

The designer can tune the PPT further if appropriate. For example, the PPT for an interrupt handler may be eliminated if that interrupt handler does not experience significant instruction cache miss effects due to context switching. PPT entries can also be altered as necessary to reflect environment, application, or software changes, and those changes applied through a simple software upgrade.

Efficiency is important with the addition of the PPT because the prefetch operations themselves add overhead — too much time spent to access the PPT and perform the prefetches will negate the gain of using Interrupt Triggered Software Prefetching. However, using tight assembly code in a scratch pad memory, an inline jump table takes only 9 instructions to perform 4 prefetches for a PPT reference (see Fig. 15). Code space is also a concern since the PPT resides in internal memory, but this implementation a PPT entry of 4 prefetches consumes only 52 bytes with the literal address.
4.3 Goals and Motivations

The intent of ITSP is to use PPTs to predict the cache miss behavior depending on the previously running task. Observations have confirmed that different task switching combinations lead to different sets of instruction cache misses, which makes sense since the previous task most recently influenced the instruction cache’s working set.

During research with ITSP, just four software prefetches triggered by an interrupt can cause a noticeable improvement in the interrupt service time with relatively low instruction overhead. This improvement is even more pronounced on real-time systems that experience a very high degree of interrupt-based context switching. These real-time systems may switch on the order of 2-20 times per ms — much more frequently than switching every 25-50 ms like programs running under Windows or Linux on a general purpose CPU [1].

![Cache misses seen during simulation.](image)

Fig. 16. Cache misses seen during simulation.
In these embedded real-time environments, the interrupt service routines are typically small, running on the order of 1000-5000 CPU instruction cycles. With cache miss penalties typically ranging between 40-80 CPU instruction cycles, even a few successful instruction prefetches can result in a useful performance improvement. Since the interrupt service routine is often a crucial region of code with tight timing requirements, this type of prefetching helps where it is needed the most.

Also as expected, it appears certain tasks interfere with the cache more than others after a context switch. Fig. 16 shows the cache misses for Task 1 when it runs after 5 different previous tasks (2–6). For each previous task, the figure shows the number of instruction cache misses for the Worst Case Execution Time (WCET) and Average Case Execution Time (ACET). In this figure, when Task 1 executes after Task 4 (the previous task), Task 1 experienced 26 cache misses during the observed worst case and 10 during the average case.

Therefore the overall goal is reduce the number of cache misses seen during WCET using the best statistical set of PPTs, given a limited amount of memory devoted to prefetching. A secondary goal is also to improve the ACET of all interrupt tasks which allows the system to run more smoothly and reduces the amount of time interrupts are disabled.
4.4 Experimental Evaluation

This section illustrates ITSP using a small embedded test system running six application benchmarks with two different cache configurations and randomized interrupt patterns. For all simulations, a commercially available ASIC simulator was used to exercise a cycle-accurate ARM processor model. The history data was collected by a Verilog testbench which also drove the simulation.

The architecture of this embedded system is again typical of many embedded systems which has already been described. A 32-bit ARM processor core [30] was used together with small 4-way set-associative instruction caches (icaches) and data caches (dcaches). A small 4KB instruction scratch pad memory holds the interrupt service kernel and the PPTs. The remaining software is stored in a cacheable SDRAM external memory model residing in the Verilog testbench. The CPU was simulated at 100 MHz with the external memory running at half that speed. On average, a cache miss penalty of 50 CPU clock cycles was seen when obtaining an entire cache line and resume operation.

The intent of these experiments was to create a multi-tasking environment using compiled C code that is typical of a real-time embedded system. For software, various routines were selected (see Table 9) from the embedded suite Mibench [36] along with the popular Dhrystone benchmark [96].
Table 9. Test simulation software for Interrupt triggered prefetching.

<table>
<thead>
<tr>
<th>Routine Name</th>
<th>Size (KB)</th>
<th>Exec Time</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Bitcounts</td>
<td>1.5</td>
<td>3</td>
<td>Count bits in a vector</td>
</tr>
<tr>
<td>2. Dijkstra</td>
<td>1</td>
<td>2</td>
<td>Shortest path</td>
</tr>
<tr>
<td>3. Arithmetic (background)</td>
<td>5</td>
<td>23</td>
<td>Various arithmetic</td>
</tr>
<tr>
<td>4. Qsort</td>
<td>.5</td>
<td>42</td>
<td>Quick sort routine</td>
</tr>
<tr>
<td>5. Rijndael AES</td>
<td>3</td>
<td>24</td>
<td>Encryption of data</td>
</tr>
<tr>
<td>6. Dhrystone (background)</td>
<td>1.5</td>
<td>7</td>
<td>Popular benchmark</td>
</tr>
</tbody>
</table>

Table 10. Foreground task interference metric comparison.

<table>
<thead>
<tr>
<th>Previous Task</th>
<th>Foreground Task</th>
<th>WCET / ACET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1.04</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1.08</td>
<td>1.03</td>
</tr>
<tr>
<td>4</td>
<td>1.09</td>
<td>1.04</td>
</tr>
<tr>
<td>5</td>
<td>1.04</td>
<td>1.04</td>
</tr>
<tr>
<td>6</td>
<td>1.07</td>
<td>1.03</td>
</tr>
</tbody>
</table>
This code was altered slightly to operate within a rapid context switching environment using the run-to-completion model. This result is a heterogeneous mix of routines that gives the processor something very different to do in each software task. The test used the Dhrystone and Mibench arithmetic tasks as time-sliced background tasks, with four other ibench tasks (tasks 1, 2, 4, 5) invoked by the foreground interrupt context. This mix gave both short (Task 6 = Drystone) and long (Task 3 = Arithmetic) background tasks, and both short (Task 1 = Bitcounts, Task 2 = Dijkstra) and long (Task 4 = Qsort, Task 5 = Rijndael) tasks.

This experimental environment allowed us to model a wide variety of task switching combinations of background tasks interwoven with foreground tasks invoked by interrupts. Interrupt arrivals were randomized with times ranging from 50 µs to 800 µs. These interrupts were equally weighted so that each foreground task was given a similar time slice, but since the run to completion times varied, the overall amount of execution time per task was different as shown in Table 9. The randomized interrupts were not intended to model a real system, but rather to cause randomized interference among tasks in order to study the effects on the instruction cache.

Overall, the total code size was 28KB. Along with the task code, an additional 15KB of code space was used for kernel and library routines that were shared among the tasks. Experiments included both 4KB and 16KB instruction caches in order to model systems experiencing different degrees of cache misses.
4.4.1 PPT Configuration

The following information was obtained during the history collection step and used to configure the PPT. This information was collected for each task switching combination:

- Overall WCET and ACET observed for that task switching combination
- Sorted list ranking the most commonly occurring instruction cache miss addresses for that task switching combination
- A complete list of icache misses during the overall WCET task switching combination

From this data, a foreground task interference metric can be used to measure the observed deviation of WCET to ACET. Table 10 shows this result as a percentage of WCET to ACET for each foreground task switching combination. Hence when Task 2 runs before Task 4, Task 4 has a worst case run time that is 18% worse than its average case run time. When Task 5 runs before Task 4, this effect is even worse.

This interference metric can be used to identify those task switching combinations that most disturb the ACET of each task. If a program contains a large number of tasks, this information can be used to select the most critical task switching combinations to improve, so as to maximize the benefit of the ITSP given the limited size of scratch pad memory.

In the experiments, 1KB of the 4KB scratch pad instruction memory was dedicated for the PPTs, leaving the remainder for the conventional interrupt service tasks. This gave sufficient space for 4 PPT tables (one for each foreground task shown in Table
9), each containing 5 entries (one for each task switching combination listed in Table 10). For each entry, four prefetch targets were chosen from the list of cache misses seen during the WCET.

The selection of the prefetch targets is a simple process. The WCET list is cross referenced against the list of most commonly occurring cache misses for that task switching combination. In this way, it was possible to choose the 4 best entries that would also help in the average case. An arbitrary selection from the WCET list can also be done in situations where there is not a strong correlation between the two lists. In summary, using prefetching to prevent a cache miss seen during a WCET scenario will help improve the situation; hence the actual prefetch target taken from the WCET list is not important. But where possible, favoritism was given to those entries appearing in the most commonly occurring list to also assist ACET situations.

4.5 Experimental Results

Following the method outlined earlier in Section 4.4.1, first simulations were run without prefetching to collect the history data and obtain baseline performance numbers. This history data was used to configure the PPTs, and the simulations were then rerun with prefetching instructions added. During this second simulation, history data was again collected to characterize the performance in the presence of ITSP. In both cases, history data was accumulated over several thousand context switches.

Fig. 17 illustrates the number of cache misses seen during the overall WCET and ACET for each task switching combination for two of the four foreground tasks. On
average, prefetching removed 3-4 cache misses each time a task was run, where on average 50 clock cycles are saved per miss. Fig. 17 also shows the effect of using a small 4KB cache on both a small task (Task 1 = Bitcounts) and a long task (Task 4 = Qsort).

Fig. 18 is similar, but shows results using smaller caches (4KB caches instead of the 16KB caches in Fig. 17) for Tasks 1 and 2. It is interesting to observe that Task 1 experiences the most interference when run after Task 4, regardless of cache size. I found similar effects for other task switching combinations, where the magnitude of the interference metric was different, but the worst case switching combination was still the same.

I summarized the results of all four foreground tasks, for the overall WCET and ACET across all previous tasks (Table 11), with and without ITSP. For WCET, the context switching combination which had the most interference was referenced. Also, the *prefetch accuracy* as well as the overall average improvement in the number of cache misses seen each time the task was run. Different PPTs were created for different cache sizes as different cache misses were observed for each environment.

Since the run times of the tasks were relatively long compared to the amount of cycles saved by the prefetching, there was not a dramatic improvement in execution time for Tasks 4 and 5. However, Tasks 1 and 2 did show modest improvement in WCET for both 4KB and 16KB cache sizes. Overall, the amount of time spent with interrupts disabled was reduced by 2% for the entire program, reflecting an overall reduction in the amount of time spent in the interrupt handlers (foreground tasks) for the entire program.
Furthermore, it was also observed that other factors caused variation in execution times, especially for one long task (Task 4 = qsort), that had fluctuations in performance due to data cache access. Although this effect was not investigated fully, Task 6 (Dhrystone) and qsort both conflicted over the data cache. Hence, when qsort ran after Dhrystone, it seemed to experience worst case execution. The Interrupt Triggered Software Prefetching still helped in this situation but was less of a dominant factor with that task. Other tasks had infrequent data cache access and thus had little influence.

Using 16KB cache sizes, an overall cache hit rate of 99.5 % was seen for the instruction cache and 99% was seen for the data cache. Most of the program was held comfortably in this cache size, resulting primarily in conflict misses rather than capacity misses. The prefetching had a fairly predictable affect on the ACET and WCET for all tasks. With 4KB caches, the hit rate was much worse — only 97% for instruction cache and 95% for data cache. It was apparent that with 4KB caches there was a high degree of conflict and capacity cache misses resulting in a stressful environment for Tasks 4 and 5. Hence the prefetching effect was diluted due to the extreme number of caches misses which dominated the performance of these tasks.

4.6 Conclusion

ITSP complements the hardware techniques described in Chapters 2 and 3 as a practical software technique to enhance efficiency during a context switch. This technique can be applied to many embedded systems that rely on commercial CPU cores with instruction caches and a small amount of scratch pad memory. No external
prefetching hardware is required for this method, which instead relies on software prefetch instructions supported by many CPU cores.

By profiling the system’s performance and collecting history data, Prefetch Prediction Tables (PPTs) can be constructed to guide the software prefetch decision. By interrogating the upcoming interrupt to reference this table, it is then possible to target cache miss sequences that are most influenced by the previously running task. Successful prefetches can help soften the blow of the inter-task interference that occurs as a result of context switching. This causes a reduction in variability of the execution time, resulting in more predictable performance for interrupt triggered real-time tasks.

Experiments indicate that this method can reduce the number of cache misses seen after an interrupt-driven context switch. Indications suggest this type of prefetching is more effective in environments which experience conflict cache misses, rather than those with tiny caches which experience conflict and capacity misses during inter-task switching. This helps improve both average and worst case execution times. Being a software technique, one possible application might be to augment other methods involving caches or scratch pad memories, targeting pathological worst case conditions not already covered by other methods.

It may be possible to enhance Interrupt Triggered Software Prefetching by adding more weight to those interrupt service routines which are higher priority or those interrupts which occur more frequently than others. Studies thus far assumed equal priorities and frequencies for all interrupts.
Fig. 17. Reduction in cache misses with prefetching using 16KB caches.

Fig. 18. Reduction in cache misses with prefetching using 4KB caches.
Compared to other techniques using software prefetch instructions, I feel this approach has advantages for real-time systems. This method makes intelligent choices for prefetch targets based on the past history of observed cache misses during an interrupt driven context switch. Without considering the inter-task interference, it is difficult to select targets for the software prefetch instructions which can help in the worst case scenarios. The technique utilized selectively chooses the best candidates for prefetching providing a performance boost with just a small number of prefetch instructions. Reducing prefetch instructions also helps reduce code space and wasteful prefetches.

The software created for embedded real-time systems is distinctly different than those programs running on traditional general purpose computers. Embedded software is specialized and largely fixed for the life of the application. Performance enhancement
can be achieved by customizing the software to boost performance in the embedded environment. These systems are constrained in terms of processing power and cache sizes so it is important for software engineers to optimize the interrupt interactions in order to arrive at the best available solution to meet critical real-time processing constraints.
Chapter 5. Dynamic Round Robin scheduling

Besides interrupts, inter-task interference was yet another key factor observed that caused undesirable CPU performance due to a cluster of cache misses. This was caused by the CPU switching to different tasks caused by interrupt driven task preemption as observed in Chapter 4. Another common source of task switching in embedded systems is time slicing of the operating system. My latest research, recently presented at DATE [13] is aimed at improving the cache performance by reducing inter-task interference. A classic way to schedule tasks in a multi-tasking embedded system is to use a round–robin CPU scheduler. This is especially true in low-cost systems that lack an operating system (OS) or use a very simple OS. While a full-featured OS such as Windows or Linux may use more complex techniques, a round-robin scheduler offers a simple and straightforward way to balance different tasks and share the CPU fairly in an embedded system [73].

In many of these embedded systems, processor cores that contain instruction and data caches are common. These processor cores keep the cache sizes small (e.g. 4K or 16K bytes) in order to reduce cost and power in the embedded environment. Cache misses are common, since the entire program cannot fit into the cache space provided, so the CPU will frequently stall while it performs external memory reads to obtain the missing cache lines. While stalls are always detrimental to the performance of the
system, the limited memory bandwidth in these embedded systems makes the cache miss
penalty particularly problematic.

Multi-tasking embedded processors present special challenges when considering
cache performance. In order to maximize the overall throughput of the system, it is
desirable to improve the performance of the caches as they service different tasks.
Though caches usually perform well while a single task is running, the situation is
different in a multi-tasking environment. Here the previously running tasks leave
memory “footprints” in the cache, so as new programs displace the ones currently in the
cache they cause a set of conflict misses that degrade the performance of the new tasks
scheduled onto the CPU [87].

The degree to which tasks interfere with each other is affected by the temporal and spatial
locality of the different tasks. Temporal locality means there is a high
possibility that recently accessed data can be accessed again in the near future. Spatial
locality refers to the tendency for adjacent memory locations to be referenced close
together in time. Tasks may also complement each other in beneficial ways for
example if they share a common library function. If the previous task fetches the
routines and stores them in the cache, which prevents a cache miss when the subsequent
task runs.

In many embedded systems, predictability is important — since embedded
software is largely stable for the life of the application; such systems are expected to have
a stable performance as well. Unfortunately, embedded systems are often more sensitive
to performance disturbances than general purpose systems, making predictability and stable performance a problem. Further, real time processing constraints are common in embedded systems where some or all tasks have processing deadlines.

Overall, much can be gained by tuning the software in an embedded environment to preserve precious CPU cycles. This builds on the observation above that tasks may complement each other in beneficial ways and allows the CPU scheduler to adaptively adjust to changes in processing load and environment in order to reduce cache misses.

5.1 Background

There are many ways to organize the software in an embedded system. How does one determine what code structure is best to avoid cache misses in a multi-tasking environment? Researchers have developed both hardware and software techniques to address this problem. This research relies on live execution profiling of the software as it runs. Profiling — a classic method introduced by Knuth — allows the behavior of the running program to be analyzed and used for feedback to optimize the performance [52].

Pettis and Hansen created an effective software technique for reordering code at the basic block level, branch alignment and procedure reordering [70]. This can be used as a compile time optimization to effectively reduce cache misses. Procedure placement using temporal ordering was improved upon by Gloy and Smith [32], using profile-driven code-placement with a weighted call graph to derive an improved procedure ordering.

Kalamatianos and Kaeli introduced a Temporal Based Procedure Reordering technique [49] that involves constructing a conflict miss graph and coloring algorithm to
produce an improved code reordering for instruction caches. As input to their algorithm they use code analysis and trace driven simulation to discover an improved layout of a program in the available memory space. Ghosh uses cache miss equations to drive compiler optimization decisions for improving cache performance [31].

Samples uses profile driven compilation in his work to help optimize code layout [80]. He describes how the software is instrumented to collect profile data. This useful feedback helps the compiler create a more optimized layout which can reduce cache misses. Beyond embedded systems, other related cache optimization topics are found with regards to loop tiling, cache performance with operating systems, and program path analysis for pre-emptive real time systems [85] [93] [91].

These software techniques all improve code performance by code analysis and evaluation of memory access patterns. The software is physically recompiled and / or linked to create a more optimal memory layout to avoid cache misses.

Hardware methods include creating cache structures more suited for reducing cache conflicts. Lee, et. al. use a temporal and spatial cache system based on time interval to optimize performance [54]. Their technique uses a small block size direct-mapped cache more suited for temporal locality and a fully associative buffer with large block size to address spatial locality.

Other techniques are software-based but more directly in support of the hardware. Yang et. al. use compiler assisted cache replacement to improve cache performance [100]. Their technique utilizes compiler hints to the memory instructions to help the
hardware make a more intelligent choice when managing the cache. The authors have shown how the problem can be stated as a 0/1 knapsack approximation that can be solved using dynamic programming.

Xia and Torrellas also use a clever means of improving cache performance by leaving software hints in the compiled code to help guide the hardware prefetching [92]. They first create a code layout more suited for temporal, spatial and loop locality based on dataflow analysis, address traces, and frequency of routine invocation. The compile time hints provide information of this optimized code layout so that the prefetching performance is more effective, thereby reducing cache misses. Interesting work was also done by Rogers, et al with regards to speculative loads [78].

This technique, which is called Dynamic Round-Robin Scheduling (DRRS), is also software-based in support of hardware. Later in this chapter, I will show how a simple adjustment to the order of tasks processed by the round-robin CPU scheduler can result in performance improvement. No changes are required to the code layout, which remains as generated by the compiler. However the round-robin schedule sequence is modified to reduce the cache misses experienced. Further, this techniques uses live execution profiling of cache misses information, so no post processing of data is required, and the program adapts dynamically as it runs.

5.2 Techniques and Implementation

After an embedded system’s software or firmware is compiled and linked, instruction addresses remain static unless the program is upgraded in the field. As a
result the methods described in the previous section can be used to organize the code in order to avoid cache misses. These methods may also be used to organize the data structures and references to the data so as to avoid data cache conflicts as well as instruction cache misses.

Dynamic Round-Robin Scheduling (DRRS) complements those techniques by monitoring and then improving the performance of a system that uses round–robin CPU task scheduling. DRRS simply reorders the round-robin scheduler’s task execution sequence. This reordering is done dynamically during program execution, and hence is a run-time improvement done in software, with no special hardware required. Like Gloy, I also realized that even small changes can make a difference in performance [33].

For example, consider the following round-robin schedule consisting of 6 tasks (see the leftmost diagram in Fig. 19). Assume the round-robin scheduler proceeds clockwise — so the task execution order is A,B,C,D,E,F and then repeats continuously in that sequence. As the tasks execute, each task replaces cache data, and a series of cache misses occur, harming performance.

Fig. 19. Two example round-robin schedules for 6 tasks.
Now suppose the execution order of talks B and C are swapped, resulting in task execution order A,C,B,D,E,F. Using a classic round-robin scheduler, the order in which the tasks are processed in each round does not matter — all that is required is that all tasks are given an equal timeslice in each round.

However, the new execution order results in a different cache usage, potentially a reduction in the number of cache misses, and thus overall better performance. This is the key to DRRS — simply changing the execution order of the tasks in the round-robin CPU scheduler to improve cache performance.

It is important to understand that DRRS does not physically change the location of the tasks in memory, which are still fixed at compile and link time. In this example, tasks B and C will still reside at the same memory locations in either schedule. The only change required to the round-robin scheduler is a small number of jump instructions added to allow any permutation of the tasks being scheduled. The change to the scheduler is described further in Section 5.3.

![Diagram](image)

Fig. 20. Running and tuning states.
DRRS continually monitors the cache performance of the system and dynamically changes the round-robin scheduler’s execution sequence. When DRRS is used, the embedded system is in one of two states — running or tuning (Fig. 20). When a system is *running*, a good round-robin schedule is in place, giving an acceptable number of cache misses and efficiently processing data. When the system detects a high level of cache misses outside an acceptable norm, it goes into the *tuning* state, and attempts to find a better schedule to reduce the number of cache misses to an acceptable level.

Note that tuning does **not** attempt to find the optimal schedule. DRRS simply tries to find a schedule that reduces the cache misses below a user-specified level. The goal of this research work was to find a low-overhead technique to provide moderate improvement, not a high-overhead technique that provides the best possible improvement.

DRRS uses a set of performance counters to measure the instruction and/or data cache misses. This is a flexible, purely-software method that requires no special hardware except for the counters, and even those could be replaced with software counters if necessary. The only OS change required to support the tuning is a small amount of code in the OS kernel to read the counters at regular intervals after some fixed number of round–robin schedules complete. These changes, along with the tuning algorithm, are described later in Section 5.4.

DRRS, like any dynamic technique, relies on some measure(s) of system performance. In this case it was decided to reduce, the number of cache misses, but the
DRRS technique could just as easily be used to reduce some other performance measure, such as execution time.

5.2.1 Motivation and Problem Statement

Embedded systems regularly undergo changes due to the environment in which they operate. For example:

- Tasks may terminate and leave the CPU schedule.
- New tasks are created and must join the CPU schedule
- Interrupts may occur and disrupt the CPU schedule
- The data set being processed by the system can change and cause the tasks to perform differently.

These changes often disturb the state of the instruction and data caches, leading to periods of cache misses. For instance, suppose static code analysis is performed and determines that the best ordering of five tasks is E,D,A,B,C. This ordering may work well under most cases, but when the system load changes and task D terminates, the remaining task order E,A,B,C may not be ideal. Or suppose a new task F is added — what is the best schedule now to minimize interference? Or suppose a data set changes, and task A now begins to use different code execution paths and data access patterns, which in turn degrades the cache behavior. In all these situations, a dynamic run-time method such as DRRS may be able to find a new task ordering to improve cache performance.

As stated earlier, just by influencing the order in which the tasks are scheduled onto the CPU, one can significantly reduce the number of cache misses experienced in
the system. This (re)ordering is done at run time and can be implemented with minimal change to the OS and scheduler and a small overall overhead. Further, the DRRS technique is completely complimentary to the other software and hardware techniques described in Section 5.2, so those methods can be used in conjunction with DRRS.

Thus the problem statement is as follows:

*Given a set of N tasks scheduled by a round-robin CPU scheduler, find a schedule order of those tasks that reduces the number of cache misses below some user-specified threshold. Adapt to system changes by adjusting the schedule order dynamically during run-time to keep the number of cache misses below this threshold.*

### 5.3 Implementation

Dynamic Round-Robin Scheduling (DRRS) begins in the *running* mode with some initial round-robin ordering. This initial state can be arbitrary or based on static analysis or other profiling technique. In the *running* mode, there is no difference in activity compared to a regular system, except that the amount of cache misses experienced are monitored at regular intervals. Thus after a set number of rounds have completed, a checkpoint is triggered to see if the number of cache misses have increased above some threshold. If so, the system enters the *tuning* state as described below and the round-robin ordering is changed to reduce the number of cache misses. The threshold could be set by a variety of methods, such as performance profiling of the system, or trace
analysis of system execution under various data sets. In this experiment for example, the
threshold is based on worst case analysis of the cache miss counters available in the
ARM CPU core. In a real-time system, execution time might need to be folded into the
threshold or checkpoint interval to meet processing deadlines. It might also be wise to
create dual thresholds with built-in hysteresis to avoid oscillations of borderline systems.
In any case, some thought needs to go into making an intelligent choice of the threshold.

Fig. 21 illustrates how the system designer might set thresholds based on CPU
performance in a real time system. In this example, the “System Critical” level means the
CPU is taking too many cycles to complete the round-robin time interval, and deadlines
are getting missed. One does not want the system to reach that point, so as soon as the
system crosses the “High Limit” threshold, it enters the tuning state. During tuning,
different round-robin schedules are tested as described below, and the system
performance eventually improves and drops below the “Low Limit” threshold. The
system then returns to the running state where is uses the new round-robin schedule. In
this illustrative example, the system tested six new round-robin sequences during successive time intervals (marked with arrows) to find a round-robin sequence that met the lower threshold.

During tuning, a new round-robin schedule order is selected by the software according to one of the method described below in Section 5.4. The system then uses the new round-robin schedule over the same checkpoint interval and compares the new performance measure to the previous best value. If there is an improvement in performance, this becomes the new best performance value and the new round-robin schedule ordering is selected. This new performance value is also compared to the threshold, and when the threshold is reached, the system moves to the running state and the new round-robin sequence is used until the next tuning cycle. The new threshold is also stored as the result of the tuning operation. Fig. 22 summarizes the activities performed during the tuning phase.

To implement the round-robin schedule order, a simple array containing an entry for each task ID to be executed is used. The array is referenced by the scheduler when the tasks are selected for CPU execution. This array is reconfigured during the Rescheduling phase shown in Fig. 22 to select the new execution sequences for the scheduler.
5.4 Algorithms Evaluation

DRRS is a run-time heuristic that attempts to iteratively improve system performance by altering the round-robin schedule order and then grading the effect of this alteration. Through a series of stepwise refinements, the number of cache misses is eventually reduced and the overall performance improves. This method is similar to many hill climbing algorithms or branch and bound heuristics; if a change to the round-robin schedule is selected that does NOT improve the performance the change is rejected. (Recall that the goal of this DRRS work was to find a low-overhead technique to provide moderate improvement, not a high-overhead technique that would guarantee the optimal schedule).
The main overhead in DRRS is in the tuning phase, where a new schedule order must be selected to test. Though DRRS would support any rescheduling algorithm in general, the following three methods were explored to generate a new schedule of tasks:

- All permutations of N tasks, exhaustive.
- Random
- Positional

With the permutation method, all combinations of the N tasks are tested in order until the performance improves and dips below the threshold. With the random method, a new schedule is selected at random and tested, continuing until the threshold is reached. The positional method will try each task in position 1 and select the best choice, then the remaining tasks in position 2, etc. until all are in the best task position.

Since this rescheduling portion of the tuning state is the main overhead added by DRRS it is important to code the rescheduling efficiently. This is accomplished by use of a simple array of N tasks which is shifted and adjusted at the checkpoint interval. Reading, comparing and maintaining the performance measures on each pass (e.g. cache miss counts) is the only other requirement for the algorithm. In this experiment, I implemented the positional rescheduling using only 40 additional instruction cycles, which are executed only in the tuning state and only once per checkpoint interval. As described earlier, the round-robin scheduler uses a scheduling array as a jump table to efficiently try different round-robin implementations, resulting in minimal overhead added to the scheduler.
Both and positional rescheduling were utilized in experiments (section 5.4). With random rescheduling, one reads a hardware random source register and used that value to reshuffle the array elements. With positional rescheduling, I used a simple inner / outer array index to juggle the array in a methodical way. Note that with only one reordering phase done per interval, 3 tasks (ABC), the 6 different orders of those tasks are tested during 6 successive checkpoint intervals: ABC, ACB, CBA, CAB, BAC, CBA. One can see how a simple swapping of the schedule array elements accomplishes this task with little execution overhead.

It was discovered in these sample experiments (see Section 5.5 below) that after a dozen or so trials, the performance usually improved enough to dip below the threshold and get back into the running state, and the main additional code overhead (the \(\sim 40\) instructions in the rescheduling algorithm, executed once per interval) did not significantly harm the performance. As in any hill-climbing algorithm, performance occasionally got worse during a test but those effects were temporary and the performance quickly improved. Overall the objective of developing a dynamic, low-overhead heuristic was reached. This method performs subtle changes to the round-robin schedule, and yet gives a worthwhile performance improvement.

5.5 Experimental Evaluation

In order to explore DRRS, I created a test workload consisting of the Dhrystone benchmark and 5 tasks selected from Mibench, an embedded systems benchmark [36]. This is similar to the test suite used in chapters 2-4. These six tasks (Table 12) were
arranged in a classic round-robin schedule where the each would be selected to run for one time interval in turn. The different tasks had different total run-times and hence required a different amount of CPU time to complete. The total executable code size including the shared libraries was 40KB.

This software was run on simulations of a commercially popular ARM11 series CPU core. This ARM CPU supports two different cache sizes for instruction and data (4K and 16K), and uses 4-way set associative caches and a selectable random or LRU replacement policy. I performed simulations using all combinations of these cache options, on a cycle accurate hardware simulator. Software code was compiled with default optimizations using the ARM Realview TM compiler tools [55].

Table 12. Test simulation software for DRRS.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>Exec.Time %</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>bitcounts</td>
<td>2%</td>
<td>Count bits in a vector</td>
</tr>
<tr>
<td>2</td>
<td>Djikstra</td>
<td>4%</td>
<td>Shortest path</td>
</tr>
<tr>
<td>3</td>
<td>Dhystone</td>
<td>21%</td>
<td>Popular Benchmark</td>
</tr>
<tr>
<td>4</td>
<td>aes</td>
<td>23%</td>
<td>Encryption of data</td>
</tr>
<tr>
<td>5</td>
<td>fft</td>
<td>40%</td>
<td>Fourier Transform</td>
</tr>
<tr>
<td>6</td>
<td>qsort</td>
<td>10%</td>
<td>Sort a set of numbers</td>
</tr>
</tbody>
</table>
Experiments began by running the system using various arbitrary schedule orderings where the number of cache misses experienced was studied. Based on these initial trials, a checkpoint interval of 10 round-robins was selected, and hence accumulated the cache miss information for both data and instruction caches. Since this was a test system without real requirements, the high threshold was arbitrarily set at the worst case observed cache misses. Then, the low threshold at 15% below this number.

Positional rescheduling was first used with the 6 tasks. Using this means, many simulations were launched using starting points above the low threshold. In all examples, compulsory misses were screened out by delaying the cache miss counts until a few round-robin iterations have elapsed. In this way, the effect of the cache footprint due to the previous running combinations and due to cold start misses was largely flushed out before measurements were taken.
Fig. 23 shows a detailed simulation result showing various round-robin schedules on the X axis with their corresponding number of instruction cache misses (in 1000’s) on the Y axis. In this example, the threshold is reached after only 4 trials so the algorithm would return to the running state at that point. However, the figure shows 10 schedules just to illustrate how positional scheduling works (i.e., testing different tasks in the first slot of the schedule, then the second slot, etc.).

As can be seen in Fig. 23, the number of instruction cache misses was reduced by altering the task ordering using positional rescheduling. However, this was not the case for data cache misses — there seemed to be no pattern of data cache miss that was influenced by positional rescheduling. Overall, the system was clearly dominated by instruction cache misses, which accounted for 95% of all cache misses. Of course, this instruction cache domination could have been due to the benchmark programs being tested, so further experimentation is needed to confirm whether or not this is a general effect.

Table 13 shows the summary of 12 simulations using both 4K and 16K caches and positional rescheduling. This table shows the starting number of cache misses when the tuning phase is entered (above the threshold) and the average number of cache misses experienced during the tuning. This average number of cache misses is important because it constitutes the cost of tuning phase. However, one can see that the cost of the rescheduling can easily be outweighed by the reduction in cache misses experienced.
Remember that on entry to the rescheduling, the system is already in a bad state; hence changes are likely to help get below the threshold.

**Random rescheduling** experiments arbitrarily juggled the elements in the scheduling array during each trial. This allowed a new round-robin schedule to be derived during each trial period with low instruction count overhead. With this trial, instead of using a threshold, all runs consisted of 20 intervals using an arbitrary random start point. The best/worst and average results achieved are shown in Table 14 and Table 15. These tables show the variation in performance that one can expect just by changing the task ordering. By selecting the best out of 20 fixed runs, it was possible to achieve as much as a 10% improvement in cache miss reduction compared to the worst case.

With 4K caches, the performance improvement was less dramatic compared to 16K caches. Most likely this difference was due to the high degree of capacity conflicts in the cache which involved frequent line flushes. This saturated the affect of cache conflict misses. The 16KB caches thus represent a more reasonable choice for this test application.
Table 13. Positional simulation result summary.

<table>
<thead>
<tr>
<th>#</th>
<th>Start/End Order</th>
<th>Start/End</th>
<th>Ave</th>
<th>Improve</th>
<th>Intervals</th>
<th>Cache Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,2,3,4,5,6, 4,2,3,1,5,6</td>
<td>8600 7542</td>
<td>7979</td>
<td>13%</td>
<td>4</td>
<td>16K</td>
</tr>
<tr>
<td>2</td>
<td>6,1,3,4,5,2 2,5,1,4,3,6</td>
<td>8878 7572</td>
<td>8072</td>
<td>15%</td>
<td>16</td>
<td>16K</td>
</tr>
<tr>
<td>3</td>
<td>4,2,6,1,5,3 2,4,5,1,6,3</td>
<td>8360 7533</td>
<td>7989</td>
<td>10%</td>
<td>7</td>
<td>16K</td>
</tr>
<tr>
<td>4</td>
<td>1,5,2,3,4,6 3,2,5,1,4,6</td>
<td>8614 7658</td>
<td>8086</td>
<td>11%</td>
<td>12</td>
<td>16K</td>
</tr>
<tr>
<td>5</td>
<td>6,5,4,3,2,1 1,4,3,5,2,6</td>
<td>9003 7663</td>
<td>8081</td>
<td>15%</td>
<td>20</td>
<td>16K</td>
</tr>
<tr>
<td>6</td>
<td>6,1,3,4,5,2 2,5,1,4,3,6</td>
<td>8878 7572</td>
<td>7894</td>
<td>14%</td>
<td>18</td>
<td>16K</td>
</tr>
<tr>
<td>7</td>
<td>3,1,2,6,5,4 1,4,6,2,5,3</td>
<td>28145 25832</td>
<td>27630</td>
<td>8%</td>
<td>21</td>
<td>4K</td>
</tr>
<tr>
<td>8</td>
<td>1,2,3,4,5,6 3,2,1,4,5,6</td>
<td>28850 26532</td>
<td>27625</td>
<td>8%</td>
<td>3</td>
<td>4K</td>
</tr>
<tr>
<td>9</td>
<td>6,5,4,3,2,1 4,3,5,6,2,1</td>
<td>27934 26695</td>
<td>27907</td>
<td>4.5%</td>
<td>19</td>
<td>4K</td>
</tr>
<tr>
<td>10</td>
<td>4,1,6,2,5,3 1,4,6,2,5,3</td>
<td>27691 26380</td>
<td>27470</td>
<td>5%</td>
<td>2</td>
<td>4K</td>
</tr>
<tr>
<td>11</td>
<td>2,5,3,6,4,1 4,5,3,6,2,1</td>
<td>27686 26575</td>
<td>27769</td>
<td>4%</td>
<td>15</td>
<td>4K</td>
</tr>
<tr>
<td>12</td>
<td>4,2,1,5,6,3 2,4,1,5,3,6</td>
<td>27983 26837</td>
<td>27586</td>
<td>4%</td>
<td>2</td>
<td>4K</td>
</tr>
</tbody>
</table>
Table 14. Prefetches experienced during simulations 16KB caches.

<table>
<thead>
<tr>
<th>Trials</th>
<th>Icache Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best Case</td>
</tr>
<tr>
<td>1</td>
<td>9088</td>
</tr>
<tr>
<td>2</td>
<td>9176</td>
</tr>
<tr>
<td>3</td>
<td>8712</td>
</tr>
<tr>
<td>4</td>
<td>9079</td>
</tr>
<tr>
<td></td>
<td>9269</td>
</tr>
<tr>
<td></td>
<td>9089</td>
</tr>
</tbody>
</table>
Table 15. Prefetches experienced during simulations, 4KB caches.

<table>
<thead>
<tr>
<th>Trials</th>
<th>Best Case</th>
<th>Average</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>26981</td>
<td>27787</td>
<td>28742</td>
</tr>
<tr>
<td>2</td>
<td>27401</td>
<td>28143</td>
<td>28984</td>
</tr>
<tr>
<td>3</td>
<td>29600</td>
<td>30187</td>
<td>30906</td>
</tr>
<tr>
<td>4</td>
<td>27400</td>
<td>28128</td>
<td>29490</td>
</tr>
<tr>
<td>5</td>
<td>29533</td>
<td>30269</td>
<td>31246</td>
</tr>
</tbody>
</table>

Icache Misses

![Graph showing Icache Misses for Trials 1 to 5]
5.6 Summary

This chapter has introduced Dynamic Round-Robin Scheduling (DRRS) as a flexible framework for improving the run-time performance of multi-tasking embedded systems. Continuous monitoring of the system performance is used to monitor the effect of the task rescheduling. With this limited experimentation, it was possible to achieve a 4% to 15% improvement in the reduction of instruction cache misses between tasks in the round-robin schedule, with low enough overhead to justify the algorithm.

Therefore results indicate that even small changes in the task ordering can result in dramatic changes in the cache performance. This can be important for multi-tasking embedded or real-time systems with delicate performance criteria. The code overhead is low and the implementation technique is flexible; which can be efficiently implemented using a task ordering array. Techniques such as DRRS can be helpful at augmenting other techniques such as code layout improvements, which are already used for embedded systems.

Much research has already been done to improve code layouts at compile and compile time to minimize cache conflict. However this improved layout works only for the given static workload, while systems follow different execution paths and workloads in the field. DRRS in contrast, monitors the cache misses and dynamically adjusts the task schedule to improve performance under changing system conditions.
Chapter 6. Conclusion

A famous computer researcher once said:

“A super computer is a device for turning CPU bound problems into IO bound problems” [9].

However embedded microprocessors are not supercomputers. They are mass produced low cost entities. Hence my definition of an embedded system is different compared to a supercomputer where I claim:

“Good embedded systems should turn IO bound problems into CPU bound problems”

What this refers to is: a perfect ideal embedded system runs with 100% CPU utilization and never has wasted cycles stalling for memory access. It is a given that the embedded microprocessor is I/O bound due to the environment which it operates. Hence one must optimize the I/O paths in and out of the system, which include the cache, in order to streamline operation and utilize the high speed potential of the processor logic, often wasted in embedded form factors.

Other issues besides memory IO performance bound the execution potential of embedded processors. Cost and time to market is a huge consideration with embedded systems. Therefore it is popular to use licensed IP cores for the CPU. As previously indicated, this limits the ability to modify the hardware contents and forces the designer to implement techniques in software or hardware external to the core. Often classic
techniques which involve instruction caches and prefetching are not sufficient or suitable to the unique environment of embedded systems.

The software running on these systems should be predictable and allow real time processing constraints to be met. Further this software is largely fixed and specialized in nature. Thus optimizing the code to suite the embedded application space is a viable strategy. This runs in contrast to general purpose computers which run a variety of different software applications often on complex operating systems. Due to time to market pressures and code size limitations embedded designers seek practical low overhead solutions to improving the embedded system. The techniques illustrated meet these goals and can be readily applied to embedded systems in the field.

The pervasiveness of embedded CPUs has enabled amazing portable devices to be created. Solving the embedded CPU performance problem is important in order for these devices to reach their full potential. This problem is compounded by conflicting design constraints placed on embedded systems, which forces designers to seek out novel approaches for design enhancements.

6.1 Contribution

This dissertation describes a flexible framework for improving the performance of embedded CPU instruction caches. A collection of techniques was developed that involve both hardware and software. Cluster Miss Prediction Prefetching (CMP) is a hardware prefetching method first developed to address the bursty nature of instruction cache misses often seen with the restrictive environment of embedded processors.
This was compared to the popular prefetch on miss technique (POM) and later combined with POM to create an even more powerful prefetch method (CMP+POM described in Chapter 3). To address the bursty behavior of cache misses due to interrupts, the Interrupt Triggered Software Prefetching (ITSP) method was developed (Chapter 4). Interference between different tasks running in a time sliced round robin also results in instruction cache misses. Thus Dynamic Round Robin Scheduling (DRRS) was created to address this concern as described in Chapter 5. Being a dynamic method, this technique goes into affect as the system is running so it can adapt to changing workloads and changing environments.

Experimentation was done with real industrial designs on cycle accurate models. Both ARM and MIPs synthesizable processor cores were used, which are the most popular processor cores used in embedded systems today. This provides a means to validate the approach using realistic performance and real-life designs, rather than pedagogical theoretical models. The internal and external memory structures supporting the CPU caches were created from actual SDRAM designs. Therefore the stalling, on-chip bus arbitration, and latency features could also be realistically modeled and precise performance numbers obtained.

Throughout this dissertation, different metrics were introduced which attempt to quantify the degree of performance degradation due to cache misses. With CMP for example, the goal is to smooth out the performance targeting hot spots in the code which cause disturbances harming the WCET. In effect, these metrics identify the cache miss
density of a set of cache misses compared to the average number of cache misses experienced as code runs. So by improving these spots, one can obtain a more predictable performance with a reduced number of cache miss bottlenecks.

Similar software sets derived from embedded processor benchmarks were used to demonstrate the usefulness across many platforms. These benchmarks were created by other researches in order to capture common functions of embedded software. With all testing, different caches sizes and configurations were used to compare and contrast how the methods worked with different design criteria. This was done to illustrate the broad scope of the technique showing it to be useful beyond the limited context of a single design framework.

Furthermore the methods described are practical and cost effective so they can be employed without a huge complexity or cost overhead. This is very important for embedded systems where time to market and cost are huge factors. Often many of the other techniques described in the literature are not suited for embedded systems. ITSP and DRRS, being software methods, do not involve any external prefetching hardware to be developed. Hence they can be employed into a system as part of a software upgrade. All of these methods can also complement other techniques for cache performance improvement which may be employed external or internal to the CPU core.

Therefore a mix of techniques are introduced in this dissertation which attack different aspects of the instruction cache performance issues, unique to embedded and real time systems. These improvements come in the form of overall system efficiency
and better worst case execution time resulting from less CPU stalling when a cache miss is experienced. In this way the designer can get the most performance out of the embedded CPU core where performance is always a premium.

6.2 Future Work

The framework introduced here creates future possibilities for additional hardware and software techniques to be developed aimed at reducing instruction cache misses in embedded systems. This includes both extensions of software and hardware techniques described in this dissertation, as well as developing novel combinations of these methods. With CMP, for example, this method can be implemented with software prefetch instructions rather than hardware prefetching. Thus when a cluster is detected, rather than feed in the instructions of the cluster miss from the Trigger Cache Line Buffer (page 28, Fig. 5.) one could feed in prefetch instructions for the upcoming cluster instead. In a multi-threaded embedded CPU, which is now becoming popular in industry [53], this would allow other useful work to happen on a different processor thread, while the prefetching transpires. Thus the blocked thread stalling for the prefetching can resume once the prefetch is done. This would smooth out the disruptions caused by clusters of instruction cache misses, and allow more work to be performed by the CPU, rather than stalling.

Cache misses occur for 3 different reasons: Capacity, Compulsory, and Conflict cache misses; also called the 3Cs. Much research has been devoted to capacity (when cache just runs out of room) and conflict cache misses (tasks have associative address
conflicts with cachelines). But little has been done in regards to compulsory misses since there is not much that can be done. These cache misses are often the cold-start misses, when the CPU starts up and has nothing in the cache. By evaluating performance of different round robin orderings using DRRS techniques, one can derive a useful start point for task orderings. Hence it is possible to initialize the round robin order on startup to minimize cache misses. Similarly, CMP could be initialized with a startup prefetching sequence, or software prefetch instructions can be added in the boot loader to minimize the affect of compulsory cache misses.

Another technique related to ITSP could be developed which uses hardware prefetching rather than software prefetching. For instance, the state of the pending interrupts could be interrogated using a hardware PPT (Table 8, page 72) to initiate prefetches. This would give the system a non intrusive early warning of upcoming performance problems which will be caused by the pending interrupts. Related to this would be a DMA based approach which loads an instruction scratch pad or layer 2 Cache (L2 cache) containing the interrupt handler and associated code. Many embedded systems today have instruction scratch pad memory and even L2 caches in addition to L1 instruction caches. A hardware based DMA working behind the scenes with the interrupt handler could intercept the interrupt, load code into the scratch pad, then pass the interrupt to the CPU once the routine is loaded into the scratch pad. In this way, the effect of cache misses caused by interrupt service routines would be eliminated, provided
The effect of cache misses caused by interrupt service routines would be eliminated, provided the scratch pad memory is big enough to hold the program. It would be easy to implement such a technique as many systems such as ARM1156 offer vectorized interrupts where an address is associated with each CPU interrupt.

A variation of DRRS can be implemented by changing the dispatch decisions in the operating system. By profiling and using history information developed by dynamically monitoring the system, once could envision a table of task orderings which result in the least amount of task interference for a given set of tasks. For example, given a set of N tasks labeled A, B, C, ..., Z. If tasks A, L, K are running, the best ordering based on previous history is K, A, L. This type of history information can be consulted when the round robin is constructed by the dispatch code. These best orders would be saved in a lookup data table referenced by the scheduler each time a time slice interval is reached.

Fig. 24. DMA based interrupt triggered hardware prefetching
Yet another variation of this DRRS technique could be called *Angry Neighbor Task Avoidance*. This technique is similar to that described above, but uses task history to make a more fine grain decision on *where* the task should belong in the round given the current arbitrary task ordering. By observing system performance, one can derive a database where tasks simply don’t get along and interfere a lot with each other. Hence when the new task say task E is joining the round robin execution, a history table is consulted indexed by this task, which helps avoid placing adjacent tasks into the round that would conflict. This dynamic approach allows run time optimizations to be made.

It would also be possible to create background task pre-emption for real-time systems. This technique uses continuous performance monitoring similar to DRRS which can be used to help a real time system which has a mixture of priority and background tasks in a round robin. For example, if a performance threshold is approached where the system may start to risk missing deadlines, currently running background tasks can be perhaps suspended and later restarted. Feedback given at fixed time intervals will help monitor the situation therefore hopefully preventing a situation where the real-time system misses deadlines.

Finally, as indicated earlier; techniques could be developed related to *data* caches, not *instruction* caches. My work here does touch on data cache effects, but not in detail. The scope of this research centers only on instruction caches. But techniques using CMP, ITSP, and DRRS could be expanded to include *data* and *instruction* cache affects. A data cache miss also causes the CPU to stall but has different behavior since data is
influenced by read/write operations which the CPU instructions execute; not the instruction flow itself. Hence conceptually the behavior of data cache misses is related to the instruction flow but only indirectly and hence different properties apply.

Overall the Cache Miss Reduction Techniques described here show promise for expansion and the potential for future research to be conducted. The growing popularity of embedded devices in the marketplace makes this a relevant design problem applicable in the industry with a wide solution space. My studies of embedded systems show that there is not single solution for the embedded CPU instruction performance problem. But rather the problem is better solved using a collection of solutions to converge on the problem. Resulting in a system more suited to meet the requirements embedded environment.
References


Workshop on Languages and Compilers for Parallel Computing (LCPC'03), October 2003.
