Volume 3, Issue 9, September 2013 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Survey on Hardware Based Advanced Technique for Cache Optimization for RISC Based System Architecture

Ghanshyam Gagged
School of Computer Science
VIT University, India

Rawat Paresh
School of Computer Science
VIT University, India

Jitendra Madarkar
School of Computer Science
VIT University, India

Abstract— This paper initially has a thorough discussion about cache and various mapping technique. Then we shift our focus on cache optimizations and discuss the motivation for doing this on the same, followed by the different optimization techniques. Further to avoid various categories of cache misses we discuss different types of cache technique to achieve higher performance [1]. Lastly we discuss few open and challenging issues faced in various cache optimization techniques.

Keywords— Cache mapping technique, Cache optimization, Cache miss, Cache Hit, Miss Penalty

I. INTRODUCTION

Cache is the smallest and fastest memory component in the hierarchy. It is aimed to bridge the gap between the fastest processor to the slowest memory components at a reasonable cost [2]. It maintains the locality of information and support the reduction of average access time. The address mapping converts physical address to the cache address.

Mapping techniques:

1. The direct mapping is the simplest one which consume less number of tag bit but it result to many block movement. In the direct mapping to accommodate k'th main memory block in to the cache, the existing block at (K mod N) location is chosen for replacement. The no. of comparator required here is one. 2. The associative memory is the fastest one but it requires more hardware, the number of comparator required is equal to the number of cache block

3. Set associative mapping contain the advantage over direct mapping and associative mapping the set size is the no. of cache block placed on it [1]. In one way set associative each set contain only one set block that is it reduces to direct mapping but if the set contain all the cache it become associative mapping.

For cache optimization we are dealing with the following techniques:

1. Reducing cache miss penalty
2. Reducing miss rate
3. Reducing hit time

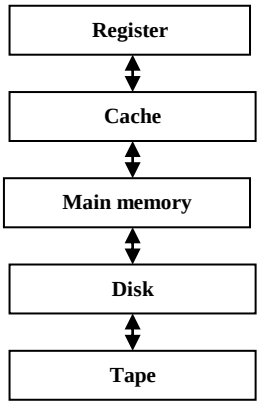

Figure 1: Memory Hierarchy

As mentioned above, a typical memory hierarchy starts with a small, expensive and relatively fast unit, called the cache, followed by a larger, less expensive, and relatively slow main memory unit. Cache and main memory are built using solid-state semiconductor material (typically CMOS transistors). It is customary to call the fast memory level the primary memory. The solid-state memory is followed by larger, less expensive, and far slower magnetic memories that consist typically of the (hard) disk and the tape [3]. It is customary to call the disk the secondary memory, while the tape is conventionally called the tertiary memory. The objective behind designing a memory hierarchy is to have a memory

system that performs as if it consists entirely of the fastest unit and whose cost is dominated by the cost of the slowest unit. The memory hierarchy can be characterized by a number of parameters. Among these parameters are the access type, capacity, cycle time, latency, bandwidth, and cost. The term access refers to the actions that physically takes place during a read or write operation. The capacity of a memory level is usually measured in bytes. The cycle time is defined as the time elapsed from the start of a read operation to the start of a subsequent read. The latency is defined as the time interval between the request for information and the access to the first bit of that information. The bandwidth provides a measure of the number of bits per second that can be accessed. The cost of a memory level is usually specified as dollars per megabytes. Figure depicts a typical memory hierarchy. The term random access refers to the fact that any access to any memory location takes the same fixed amount of time regardless of the actual memory location and/or the sequence of accesses that takes place [1]. For example, if a write operation to memory location 100 takes 15 ns and if this operation is followed by a read operation to memory location 3000, then the latter operation will also take 15 ns. This is to be compared to sequential access in which if access to location 100 takes 500 ns, and if a consecutive access to location 101 takes 505 ns, then it is expected that an access to location 300 may take 1500 ns. This is because the memory has to cycle through locations 100 to 300, with each location requiring 5 ns.

II. DETAILED PROBLEM DEFINITION

- Reducing Cache miss penalty
  Cache miss penalty can be reduced by following techniques [4]:

a. Multilevel caches

The performance gap between processors and memory leads the architect to this question: Should I make the cache faster to keep pace with the speed of CPUs, or make the cache larger to overcome the widening gap between the CPU and main memory?

Although the concept of adding another level in the hierarchy is straightforward, it complicates performance analysis. Definitions for a second level of cache are not always straightforward. The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high [3]. This observation inspires design of huge second level caches—the size of main memory in older computers! One question is whether set associativity makes more sense for second-level caches.

b. Critical words first and early restart

Multilevel caches require extra hardware to reduce miss penalty, but not this second technique. It is based on the observation that the CPU normally needs just one word of the block at a time [4]. This strategy is impatience: Don't wait for the full block to be loaded before sending the requested word and restarting the CPU. Here are two specific strategies:

Critical word first-Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block. Critical-word-first fetch is also called wrapped fetch and requested word first.

Early restart-Fetch the words in normal order, but as soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution.

c. Giving priority to read misses over writes

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

The cost of writes by the processor in a write-back cache can also be reduced [5]. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and then write memory. This way the CPU read, for which the processor is probably waiting, will finish sooner. Similar to the situation above, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

d. Merging write buffer

If the buffer is full and there is no address match, the cache (and CPU) must wait until the buffer has an empty entry. This optimization uses the memory more efficiently since multiword writes are usually faster than writes performed one word at a time. The optimization also reduces stalls due to the write buffer being full. Assume we had four entries in the write buffer, and each entry could hold four 64-bit words [3]. Without this optimization, four stores to sequential addresses would fill the buffer at one word per entry, even though these four words when merged exactly fit within a single entry of the write buffer.

e. Victim Caches

One approach to lower miss penalty is to remember what was discarded in case it is needed again. Since the discarded data has already been fetched, it can be used again at small cost. Such "recycling" requires a small, fully associative cache between a cache and its refill path. Figure 5.13 shows the organization. This *victim cache* contains only

blocks that are discarded from a cache because of a miss—"victims"—and are checked on a miss to see if they have the desired data before going to the next lower-level memory. If it is found there, the victim block and cache block are swapped.

- Reducing miss rate

  Miss rate can be reduced by following technique:

a. Using larger block size

The simplest way to reduce miss rate is to increase the block size. Larger block sizes will reduce compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

b. Using larger caches

The obvious drawback is longer hit time and higher cost. This technique has been especially popular in off-chip caches: The size of second or third level caches in 2001 equals the size of main memory in desktop computers from the first edition of this book, only a decade before.

c. Higher associativity

The first is that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative [4]. You can see the difference by comparing the 8-way entries to the capacity miss column, since capacity misses are calculated using fully associative cache. The second observation, called the *2:1 cache rule of thumb* and found on the front inside cover, is that a direct-mapped cache of size *N* has about the same miss rate as a 2-way set associative cache of size *N*/2. This held for cache sizes less than 128 KB.

d. Way prediction and pseudo associative caches

Another approach reduces conflict misses and yet maintains the hit speed of direct mapped cache. In *way-prediction*, extra bits are kept in the cache to predict the set of the *next* cache access [4]. This prediction means the multiplexor is set early to select the desired set, and only a single tag comparison is performed that clock cycle. A miss results in checking the other sets for matches in subsequent clock cycles.

e. Compiler optimizations

Thus far our techniques to reduce misses have required changes to or additions to the hardware: larger blocks, larger caches, higher associativity, or pseudo-associativity. This final technique reduces miss rates without any hardware changes.

This magical reduction comes from optimized software—the hardware designer's favourite solution! The increasing performance gap between processors and main memory has inspired compiler writers to scrutinize the memory hierarchy to see if compile time optimizations can improve performance. Once again research is split between improvements in instruction misses and improvements in data misses.

- Reducing hit time

  Hit time can be reduced using following technique:

a. Small and simple cache

A time-consuming portion of a cache hit is using the index portion of the address to read the tag memory and then compare it to the address. It is also critical to keep the cache small enough to fit on the same chip as the processor to avoid the time penalty of going off-chip. The second suggestion is to keep the cache simple, such as using direct mapping [5]. A main benefit of direct-mapped caches is that the designer can overlap the tag check with the transmission of the data. This effectively reduces hit time.

b. Avoiding address translation during indexing of the cache

The guideline of making the common case fast suggests that we use virtual addresses for the cache, since hits are much more common than misses. Such caches are termed *virtual caches*, with *physical cache* used to identify the traditional cache that uses physical addresses. Page level protection is checked as part of the virtual to physical address translation, and it must be enforced no matter what. Another reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed [6]. A third reason why virtual caches are not more popular is that operating systems and user programs may use two different virtual addresses for the same physical address. One alternative to get the best of both virtual and physical caches is to use part of the page offset-the part that is identical in both virtual and physical addresses-to index the cache.

c. Pipelined cache access

The final technique is simply to pipeline cache access so that the effective latency of a first level cache hit can be multiple clock cycles, giving fast cycle time and slow hits. For example, the pipeline for the Pentium takes one clock cycle to access the instruction cache, for the Pentium Pro through Pentium III it takes two clocks, and for the Pentium 4 it

takes four clocks. This split increases the number of pipeline stages, leading to greater penalty on mispredicted branches and more clock cycles between the issue of the load and the use of the data.

d. Trace cache

A challenge in the effort to find instruction level parallelism beyond four instructions per cycle is to supply enough instructions every cycle without dependencies. Trace caches have much more complicated address mapping mechanisms, as the addresses are no longer aligned to power of 2 multiple of the word size [6]. They have other benefits for utilization of the data portion of the instruction cache. Trace caches store instructions only from the branch entry point to the exit of the trace.

III. RESULTS

Table I: Comparison table for cache optimization techniques

	Technique	MR	MP	HT	Complexity
Miss rate	Larger Block Size	+	-		0
	Higher Associativity	+		-	1
	Larger caches	+			2
	Way prediction and pseudo associative caches	+			2
	Compiler optimizations	+			3
Miss penalty	Multilevel caches		+		1
	Critical words first and early restart		+		2
	Giving priority to read misses over writes		+	+	2
	Merging write buffer		+		3
	Victim Caches		+		1
Hit time	Small and simple cache	-		+	0
	Avoiding address translation during indexing of the cache			+	2
	Pipelined cache access			+	1
	Trace cache			+	2

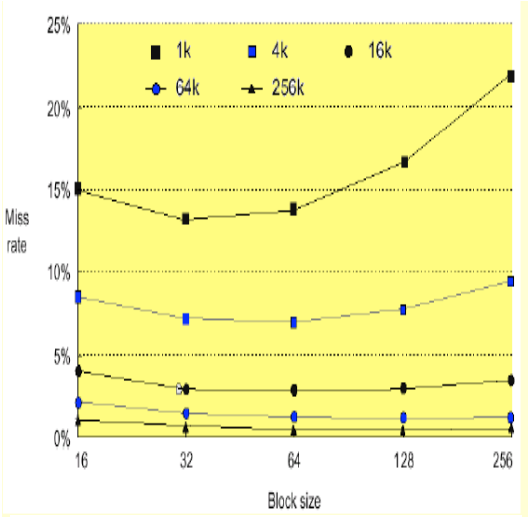

Figure 2: Reduce Misses via Larger Block Size

Larger block sizes reduce compulsory misses (principle of spatial locality) [5]. Conflict misses increase for larger block sizes since cache has fewer blocks. The miss penalty usually outweighs the decrease in the miss rate making large block sizes less favored.

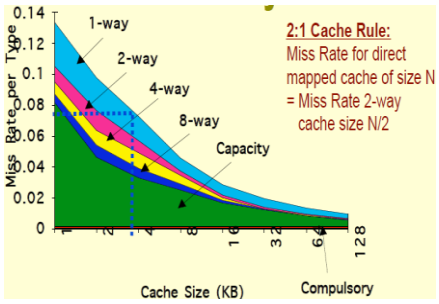

Figure 3: Reduce Misses via Higher Associativity

Greater associativity comes at the expense of larger hit access time [6]. Hardware complexity grows for high associativity and clock cycle increases.

IV. CONCLUSION

Although the loop-cache provided less significant energy savings, many embedded processor does include an on-chip loop-cache to avoid the power hungry off-chip accesses without incurring the area/energy penalty of a traditional instruction cache. Overall, smaller loop-cache sizes of the order of 256 to 512-bytes provided the maximum energy savings. For smaller loop-cache sizes, the dynamic scheme clearly out-performs the static scheme, although for larger sizes, static seems to be an attractive option. For the partitioned data cache, the four partition configuration provided the maximum energy savings. This provides just sufficient associativity to reduce the conflict misses, while allowing the compiler enough flexibility in isolating accesses to different partitions so as to pro-actively reduce conflicts while eliminating redundant cache accesses. Scaling to eight partitions actually resulted in reduced energy savings with higher area overhead.

To conclude, all three schemes must be collectively applied to reduce the overall memory power. Depending on the available area budget, the sizes of each of the hardware structures can be varied. Given a processor with one or more of these features, the compiler techniques developed in this dissertation can be used to effectively utilize this storage structures to save energy and also to improve performance.

V. ACKNOWLEDGMENT

The authors would like to thank the School of Computer Science and Engineering, VIT University, for giving them the opportunity to carry out this project and also for providing them with the requisite resources and infrastructure for carrying out the research.

REFERENCES

[1] "Aware Cache Optimization Techniques for Multi Core Processors"**,** Hasina Khatoon, Shahid Hafeez Mirza, Talat Altaf; 2011 Frontiers of Information Technology.

[2] "Memory Hierarchies-Basic Design and Optimization Techniques", Rahul Sawant, Bharath H. Ramaprasad, Sushrut Govindwar, Neelima Mothe; Survey Paper University of Massachusetts, Dartmouth.

[3] "An Overview of Cache Optimization Techniques and Cache", Markus Kowarschik and Christian Wei; Fakultat fur Informatik Technische University at Munchen, Germany.

[4] "A Trace Cache Microarchitecture and Evaluation", Eric Rotenberg, Steve Bennett; IEEE TRANSACTIONS ON COMPUTERS, VOL. 48, NO. 2, FEBRUARY 1999

[5] "Investigation of Impact of Victim Cache and Victim Tracer on a Fully Associative Disk Cache", R. Pendse, N. Kushanagar, U. Walterscheidt; Department of Electrical Engineering, The Wichita State University, 1845 North Fairmount

[6] "David A. Patterson, John L. Hennessy. "Computer organization and design: the hardware/software interface". 2009. ISBN 0-12-374493-8, ISBN 978-0-12-374493-7 "Chapter 5: Large and Fast: Exploiting the Memory Hierarchy". p. 484."

