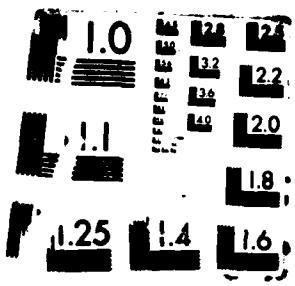
AD-A181 572 ON-CHIP INSTRUCTION CACHES FOR HIGH PERFORMANCE PROCESSORS(U) STANFORD UNIV CA COMPUTER SYSTEMS LAB A AGARWAL ET AL. 1987 MDA903-83-C-0335 1/1

UNCLASSIFIED F/G 12/7 NL


1.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
1.1
1.25
1.4
1.6

Advanced Research in VLSI
Proceedings of the 1987
Stanford Conference

The MIT Press
Cambridge, Massachusetts

On-chip Instruction Caches for High Performance Processors

Anant Agarwal, Paul Chow, Mark Horowitz,
John Acken, Arturo Salz, and John Hennessy

Computer Systems Laboratory
Stanford, CA 94305

1 Abstract

Continued increases in clock rates of VLSI processors demand a reduction in the frequency of expensive off-chip memory references. Without such a reduction, the chip crossing time and the constraints of external logic will severely impact the clock cycle. By absorbing a large fraction of instruction references, on-chip caches substantially reduce off-chip communication. Minimizing the average instruction access time with a limited silicon budget requires careful analysis of both cache architecture and implementation. This paper examines some important design issues and tradeoffs that maximize the performance of on-chip instruction caches, while retaining implementation ease. Our discussion focuses on the instruction cache design for MIPS-X, a pipelined, 32-bit, reduced instruction set, 20 MIPS peak, CMOS processor designed at Stanford. The on-chip instruction cache is 2K bytes and allows single-cycle instruction accesses. Trace driven simulations show that the cache has an average miss rate of 12% resulting in an average instruction access time of 1.24 cycles. Reprints

2 Introduction

With the rapid improvement in processor architecture, led by the RISC processors, and with advances in VLSI technology, the cost of off-chip communication has not kept pace with improvements in the clock rates of VLSI processors. Consequently, the performance of current high-performance VLSI processors is memory bandwidth limited. Including memory on the processor chip to reduce the cost of memory accesses becomes imperative to attain higher performance [12,19,11].

cost of increasing the miss service time. Moreover, both the miss rate and the traffic ratio do not consider implementation. This omission is significant because, as we show later, performance is often more sensitive to the implementation than to the cache architecture.

In this paper we concentrate on the average instruction access time which depends both on the miss rate and the miss penalty. If the cache miss rate is m, and the miss service time is T_{miss} cycles, average instruction access time, $T_{acc} = 1 + mT_{miss}$ cycles; we assume an instruction access takes one cycle if it is present in the instruction cache.

The cache parameters of interest include the cache size, number of sets (or rows) set size, block size, sub-block size, and replacement algorithm [20]. Write policies are not relevant to us because we disallow writes into the instruction stream. The set size or degree of associativity is the scope of associative search. Block size or line size is the amount of storage associated with an address tag. A sub-block (or transfer block [11]) is the portion of a block transferred from memory on a cache miss. Since a block can simultaneously have invalid sub blocks in addition to valid ones, each sub block must have a valid bit associated with it. The replacement algorithm is the process used to select one of the blocks in a given set for occupation by a newly referenced block.

Cache size is limited by the amount of chip space available for both storage of instructions and the address tags. The choice of cache parameters depends on a number of factors including (1) the miss rate achievable, (2) the timing of a cache access and how it fits in with the timing of the rest of the machine, and (3) implementation ease.

3.2 Evaluation methodology

Initially, we investigate the miss rates of various cache organizations. Then, from an implementation standpoint, we analyze the cache access timing and the miss penalty associated with each organization. Finally, we determine the average instruction access time.

Trace driven simulation (TDS) is used to obtain the cache miss rates. Because of its flexibility and ease of use, TDS is a popular technique for cache performance evaluation [20,2]; however, TDS does have some drawbacks. It may not be as accurate as hardware measurement because traces seldom reflect true workload behavior. TDS results are often optimistic because large applications, usually with poor cache performance, are hard to trace; moreover, the effect of multiprogramming, another cause of cache performance degradation, is hard to include in TDS studies.

Fortunately, these problems are not very serious in studying small instruction caches. Since small caches self purge after a few thousand references, multiprogramming has little effect on performance. Even simple models for multiprogramming, such as starting with an empty cache every few thousand references, are sufficient. In some cases (e.g., MIPS-X) this simple model can be an accurate representation of an actual virtual address cache, where the cache is flushed every time a new user process is started. Our initial analyses ignore the effects of multiprogramming. Later, we use the cache flushing model to study the impact of context switching.

In the initial phases of the MIPS-X processor design we did not have either a running software system or an instruction simulator for MIPS-X. Much of the design was based on MIPS traces (MIPS [8] is the predecessor to MIPS-X) assuming a similar behavioral trend for MIPS-X. The MIPS-X software system and a simulator have since been developed and we have corroborated our earlier findings, with the only difference being that the MIPS-X cache performance turned out to be slightly worse than predicted because *MIPS-X code is less* dense, and our current benchmarks are larger.

In this paper, we present the results using large MIPS-X benchmark traces obtained from a MIPS-X instruction level simulator. Ten Pascal (P) and Lisp (L) benchmarks are used:

Bigfm	**Fiduccia-Mattheyes graph partitioning algorithm (P).**
Dnf	**Converts logic equations to disjunctive normal form (P).**
Hopt	**A simple global optimiser for Pascal (P).**
Simu	**Operating system paging simulator (P).**
Upas	**Pascal compiler front end (P).**
Comp	**First pass of the PSL compiler (macro expand) (L).**
Frl	**Frame representation language (L).**
Gc	**Deduction with garbage collection (L).**
Rat	**Rational expression evaluator (L).**
Opt	**Compiler - data flow and optimization (L).**

These programs are fairly large, ranging from 50,000 to 800,000 bytes in static code size, and we feel they provide realistic cache performance numbers.

4 Cache Organization

This section discusses tradeoffs in the selection of cache parameters including number of sets, set size, and block size. The basic question is how to best utilise a given amount of chip area of a particular aspect ratio to obtain the maximum performance. Early cache studies (e.g., Strecker [22], and Smith [20]) compared cache performance for various cache organizations assuming a fixed amount of data storage. Alpert [3] stressed that the total circuit area associated with the cache, including both tags and data, must be considered for integrated microprocessor caches. Our experience shows that in addition to area, the aspect ratio is also important.

To reasonably limit the number of variables, we explore the design space available for the MIPS-X processor. We encourage the reader to concentrate on the evaluation procedure rather than on the final result, which may well be different given other basic constraints. Although the total cache size is fixed, a wide variety of organizations exploiting various features of program behavior are possible. We will consider in turn a conventional cache or a c-cache, a buffer, and what we call a hybrid buffer.

A *c-cache*, for the purpose of this analysis, is an organization that uses about half the available memory space for the tags. Each block consists of one to four words (each word is 4 bytes). A possible 512-word c-cache organization could have 512 sets (rows), associativity one, and block size one word. A portion of the address first indexes into a cache set, followed by an associative tag compare against the blocks in that set.

A *buffer* is a set of a few large blocks, block size being eight words or more. For example, a 512-word buffer could be organized with set size eight, and block size 64 words. Since a buffer has only one set, the associative search can be started without the indexing operation. This organization has the minimum number of tags.

A *hybrid buffer* is also investigated because the more straightforward organizations typically used in instruction buffers (e.g., CRAY-1 [5]), or in previous *VLSI* processor instruction caches (e.g., Motorola MC68020 [15], Zilog Z80,000 [3]), are not optimal in our case. As the name suggests, a hybrid buffer has the features of both a c-cache and a buffer. Like a buffer it has a large block size and *few* tags; consequently the tag store is small. It is similar to a c-cache and differs from a buffer because it has more than one set, typically two or four. A typical 512-word hybrid buffer could have two sets, set size 16, and block size 16 words.

We assume a sub-block size of one word. In other words, a single instruction is fetched into the cache on a miss.[1] Each word in the cache or buffer is associated with a valid bit. The replacement algorithm (discussed in detail later) is pseudo-random.

4.1 Technology Constraints

The technology, organization, and performance of MIPS-X greatly constrained the cache design. The most important constraint was size. Roughly half of the interior die area was allocated to the cache, giving it a space of 4mm by 6mm. The floorplan of the processor fixed the aspect ratio. The datapath was expected to be at least 6mm long and the cache had to fit above the datapath. In our design rules, the area of a 6 transistor static memory cell was 30μm by 40μm, which allowed us to build roughly a 16K-bit memory in the available area. We considered using dynamic memory cells, but decided it was not worth the technology risk.

From preliminary design and layouts for the sense-amplifiers, column multiplexers and tag compare logic associated with the cache, the column multiplexers and the sense-amplifiers were estimated to require around 300μm of space below the cache RAM array, and the tag logic an additional 300μm. The large space for the tag logic was a result of the 24 wires needed to provide the comparison address for the tags. The wire pitch was about 10μm (second-level metal).

In addition to the area constraints, MIPS-X also constrained the access time of the cache; an instruction fetch had to complete in a single 50ns clock cycle. To meet the cycle time constraint, we felt the cache would only have time for a single RAM access per cache access. Thus, we were concerned about implementing a set-associative cache since we would need to first fetch the tags and then fetch the correct data word. To alleviate the need for sequential fetches, we organized the cache so that the tag information could arrive late in the cycle. The tags were only used for the column decode. In this organization, when the word-line rose, all possible data words were fetched onto the bit-lines. The tag information was used to select the correct set of bit-lines to the sense amplifiers. The delay from the tag becoming valid to output valid was short since it bypassed the delay incurred in driving the bit-lines. The limitation of this technique was that it

[1] Many papers support sub-block placement in small on-chip caches [3,11,13].

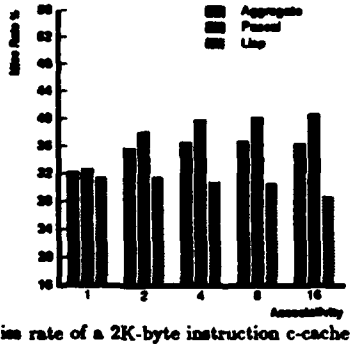


Figure 1: Miss rate of a 2K-byte instruction c-cache with block size 4 bytes. The aggregate miss rates are calculated as the average miss rate of all benchmarks weighted by the number of references.

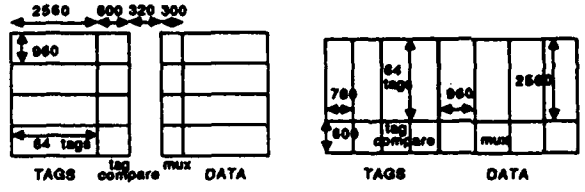
required 32 bit-lines for each degree of associativity used. A 4-way set-associative cache would need 128 pairs of bit-lines.

The cache designs described below attempt to meet both the size and organizational constraints described in this section.

4.2 C-cache

For the c-cache study we initially use a block size of one word (4 bytes). Therefore, half the area is taken up by the tags; the data store consists of 256 words. To reduce the area required by the tag store, we also try caches with a larger cache block size. We assume that if the block size is greater than two words, we can squeeze in 512 words for instructions.

Figure 1 shows the miss rate for an instruction c-cache with associativity ranging from one through 16. The number of sets correspondingly range from 256 to 16. The best possible aggregate miss rate of 32.26%, interestingly enough, is achieved by a direct mapped c-cache. We felt that some anomaly in the replacement scheme (pseudo-random) caused the miss rate to go up with associativity for the small cache. A later simulation with LRU replacement also showed this behavior for all the Pascal benchmarks with the exception of Upas. This behavior can be explained by examining the reference and collision pattern in a small cache, and is further discussed in [21,1]. Lisp



(a) Size: 6040 by 3840 μm. (b) Size: 6960 by 3160 μm.

Figure 2: Floorplans of a c-cache with 1K bytes of data store. Number of sets is 64, associativity is four, and block size is four bytes; all dimensions are in μm.

benchmarks do not show this anomalous behavior to the same extent as Pascal, because Lisp tends to have a higher frequency of procedure calls and shorter bodies of sequential code. This causes an increased probability of interference that can be reduced by associativity.

Possible floorplans for a four-way set-associative c-cache are shown in Figure 2. Note that lesser associativity caches can use the same basic floorplan. The actual dimensions are slightly greater than shown in the figure when the precharge and decode logic is added. Although both arrangements have the same area, only the former was suitable to our purpose due to the aspect ratio constraint.

We have assumed, thus far, that half the area is occupied by tags. Other c-cache types with larger block sizes and a fewer number of tags are also possible. For instance, a c-cache with a block size of four would have the tags occupying only a fourth as much area as the data portion. For these large block sizes we thought that we might be able to squeeze in 2K bytes of data (the same as for a buffer scheme) with the corresponding tags.

The miss rates for block sizes of 4, 16, and 32 bytes are given in Table 1. Because a full 2K bytes of instructions are stored, the miss rate is reduced substantially for the larger block sizes. A c-cache with 16 sets, set size 8, and block size 16 bytes achieves the lowest miss rate of 20.96%. Unfortunately, the combined area occupied by the tags and data (see Figure 3) is 4420 by 7680 μm, which exceeds the space we had available.

Clearly, for caches as small as 2K bytes, size is the single most important parameter affecting the miss rate and other parameters

Block Size	Data Size	Miss Rate (%)			
		D=1	D=2	D=4	D=8
4 bytes	1K bytes	32.26	35.80	36.67	36.82
16 bytes	2K bytes	23.28	22.16	21.89	20.96
32 bytes	2K bytes	24.91	23.65	22.96	21.90

Table 1: Effect of large cache-block size. D is degree of associativity.

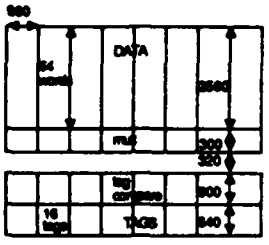


Figure 3: Floorplan of a c-cache with 2K bytes of data store, set size 8, and block size 16 bytes. Size: 7680 by 4420 μm.

play only a secondary role; e.g., doubling the associativity for the 2K byte cache with block size 16 bytes, causes the miss rate to decrease slightly from 23.28% to 22.16%; and doubling the block size causes the miss rate to marginally increase from 23.28% to 24.91%. Some parameters, however, affect the cache timing. For instance, an associative cache requires tag access, compare, a select and a drive; while a direct-mapped cache requires just the access, compare, and drive.

4.3 Buffer

Alpert [3] showed that reducing the number of tags is desirable when the area available for the cache is small and fixed. Our experiments show that this is true even to a greater extent for instruction caches. Instructions tend to show high spatial locality, which large block sizes can effectively exploit. Large block sizes are particularly attractive for reduced instruction set computers because of poorer code density and the correspondingly larger basic block sizes in the code. Buffers are attractive because they minimize the number of tags, and have the added advantage that they are well suited to prefetch schemes such as

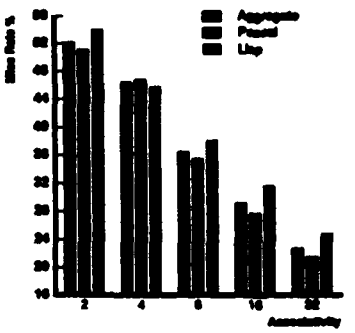


Figure 4: Miss rate of a 2K-byte instruction buffer.

load forward,[2] as well as the MIPS-X prefetch scheme outlined later.

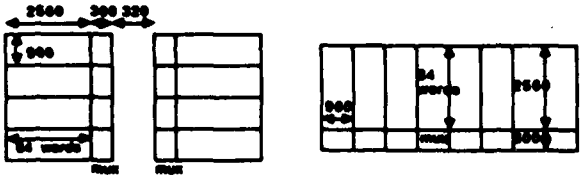
Most of the area is used for storing instructions in a buffer; hence, we do not show tags in our buffer floorplans. Access time can be made lower by (1) eliminating the indexing operation to choose a set, and (2) decreasing the tag access time by using fast circuits[3] and placing the few tags and the valid bits close to the address generation logic. For example, in MIPS-X the tags are located in the datapath itself.

We studied instruction buffers ranging from a block size of 1024 bytes and associativity two to a block size of 64 bytes and associativity 32. Figure 4 shows the miss rates. The most striking feature of the graph is the importance of block size. Smaller block sizes allow a larger associativity. Clearly, the miss rate can be very high for large block sizes. The reason is that for low associativity and a correspondingly large block size, major portions of blocks tend to be left unused. The lowest miss rate is 22.79% for a 32-way set-associative buffer, which is substantially lower than that for a direct mapped c-cache, and very similar to the best miss rate achieved for a c-cache (20.96%). It achieves this miss rate at a smaller silicon area than the comparable c-cache because it uses a smaller number of tags.

A buffer is more effective for Pascal benchmarks than for Lisp benchmarks, while the opposite is true for a c-cache organization. Because Lisp code has on average shorter bodies of sequential code

[2] Load forward was implemented in [6] and also studied by Hill and Smith [13].

[3] Although faster circuits are larger and consume more power, the overall size and power increase is small because of fewer tags.



(a) Size: 6040 by 3840 μm. (b) Size: 7680 by 2880 μm.

Figure 5: Floorplans of a buffer with set size 8, block size 256 bytes.

than Pascal, large buffer blocks tend to be under-utilized resulting in poorer cache performance for Lisp. Empirical evidence of this behavior is given by the average number of words utilized per block in a fully-associative 2K-byte buffer with block size 16 words for both Pascal and Lisp. The average block utilizations by Pascal and Lisp benchmarks are 10.9 and 8.8 respectively.

Possible floorplans for the various buffers are given in Figure 5. A set size of 8 is implementable using the layout shown in Figure 5(a). The layout shown in Figure 5(b) is too long in one dimension. Unfortunately, larger set sizes required to achieve reasonable performance are hard to implement. Figure 6 shows the floorplan for an associativity of 16. The layout is wider because of the extra bus routing channels required. When the area required by the decode and precharge logic for each of the banks is added, the dimensions along the width become much larger than we can allow.

4.4 Hybrid Buffer

A c-cache suffers from the drawbacks that tags occupy valuable space and tag access requires a RAM access. A buffer reduces these problems by reducing the number of tags and using special structures to reduce the effective tag access time. A pure buffer requires a high degree of associativity, making the actual RAM harder to design (it needs to have a large number of bit lines). Since the large associativity is required only to keep the block size down, we will provide the same block size with a lower associativity in a structure called a hybrid buffer.

A hybrid buffer simulates a higher associativity in the following manner. Consider two regular buffers where instructions map to either

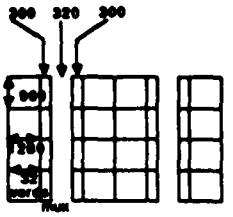


Figure 6: Buffer with set size 16 and block size 128 bytes. Size: 6960 by 3840 μm.

one of the buffers depending on the value of a bit in the address. This is similar to a c-cache with two sets indexed by a bit in the address. If each set (or buffer) has equal probability of being the target of a block, the number of instructions that will fall into any one buffer is halved, which effectively doubles the available associativity. Thus, this scheme provides the benefits of a higher associativity without implementation problems. For example, the miss rate for a hybrid buffer with two sets and associativity 16 is 22.48% which is very close to 22.79% for a buffer with associativity 32.

From an implementation point of view, associativity of 16 is still hard to achieve. A hybrid buffer with associativity 8 is implementable; unfortunately the miss rate increases from 22% to 26%. The solution is to extend the number of sets to four with the assumption that the probability distribution of blocks into the four sets is uniform. Two bits are now used to index into one of four sets. The miss rates for a hybrid buffer with four sets are plotted in Figure 7.

In this case the miss rate for a hybrid buffer with degree of associativity eight is slightly worse than for the buffer with associativity 32: 23.17% compared to 22.79%. The difference is because our assumption that each set is equally likely to be the target of a block is not quite true. The variance in the number of instructions that map to any one set causes a higher number of misses.

A floorplan for an eight-way set-associative hybrid buffer is shown in Figure 8. Two bits index into one of four sets, four bits form the block offset, and the rest of the word address forms the tag. The size of 3840 by 6040 μm can comfortably fit in the available silicon area.

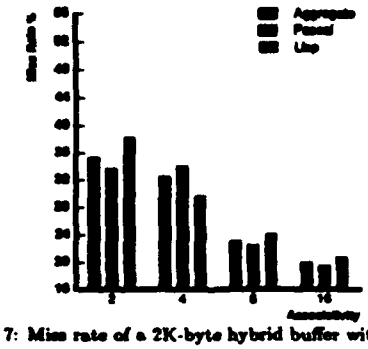


Figure 7: Miss rate of a 2K-byte hybrid buffer with four sets.

4.5 Timing Considerations

Now that we have determined several possible cache organizations based on their hit ratios and on their physical layouts, we must examine how they fit in with the timing of the rest of the machine. The only timing specification used so far has been that the cache access must be within the 50 ns cycle time of the machine. The other important timing consideration is how cache misses can be handled. To understand how various cache organizations affect the instruction cache miss timing of MIPS-X, we will first describe the MIPS-X pipeline.[4] Then we will show how the timing of the cache hit detection affects the number of cycles needed to service a cache miss. We will also use these timings to show how two instructions can be fetched back on a cache miss to almost halve the miss rate.

4.5.1 The MIPS-X Pipeline

MIPS-X is heavily pipelined so that one instruction can be issued every 50 ns. Each instruction is divided into five pipeline stages and each stage is divided into two 25 ns phases called ϕ_1 and ϕ_2. The pipeline stages and their functions are described in Figure 9. The pipeline is conceptually easier to understand if you think of an additional stage called IF_{-1} that occurs before the IF stage. During IF_{-1}, the Program Counter unit generates the address of the instruction to

[4] A detailed discussion is presented in [14,6].

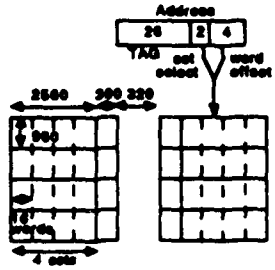


Figure 8: A set size 8 hybrid buffer floorplan. Size: 6040 by 3840 μm.

be fetched on the following IF. To denote an inserted cache miss cycle we use CM_n, where n is the number of the cache miss cycle.

4.5.2 Instruction Cache Miss Timing

Hit detection timing affects the number of cycles needed to service an instruction miss in MIPS-X. The short cycle time, coupled with the necessary chip crossings, means that external memory fetches take longer than one cycle. The address must be presented to the processor pads sufficiently early to ensure it is valid on the external pins by the time ϕ_2 falls (see Figure 9). Therefore, the datapath must drive the address bus early in ϕ_1 to start a memory fetch on the following cycle. The external cache memory then drives the processor pins with the required word by the end of ϕ_2 of the following cycle. Thus, a memory access takes one and a half cycles from the time the address is computed.

For the c-cache configurations, the tags and tag comparison have to be put in the cache array to make the implementation feasible. This means the critical path for miss detection involves driving the address up to the cache array, fetching the tags, doing the comparison, and then getting the hit or miss result back to the datapath. Referring to Figure 10, the instruction address is generated during ϕ_2 of IF_{-1}, driven to the cache array during ϕ_1 of IF, the tag fetch begins late in ϕ_1 of IF, and the comparison is computed and driven to the datapath during ϕ_2 of IF, too late to start an external fetch in the same cycle.

If the number of tags is small, as in the buffer or hybrid buffer schemes, the tags can actually be placed in the datapath close to the

IF_{-1}	ϕ_1	Compute PC
	ϕ_2	Drive PC onto PCBus
IF	ϕ_1	
	ϕ_2	Instruction fetch
RF	ϕ_1	Instruction and register decode
	ϕ_2	Register fetch
ALU	ϕ_1	ALU or shift operation for compute instructions; address computation for load or store instructions; destination and condition computation for branches
	ϕ_2	Drive address to pads for load or store
MEM	ϕ_1	Wait for external cache memory access
	ϕ_2	Drive data to pads for stores; Data arrives at pads late in this phase for loads
WB	ϕ_1	Write result into the destination register
	ϕ_2	

Figure 9: MIPS-X pipeline stages.

PC unit, making hit or miss detection in the tags much quicker. Tag compare is simply achieved by doing an associative compare of the tags to the PC bus and "OR-ing" all the match lines. However, the critical path for miss detection still involves driving the address to the cache array to fetch the valid bits.[5] Therefore, as for the c-cache, the miss signal arrives at the datapath by the end of ϕ_2 of IF.

The miss penalty for the c-cache or buffer schemes with valid bits stored in the cache array is three cycles. The timing of an instruction cache miss for the c-cache is shown in Figure 10. Because the miss signal arrives at the datapath late in ϕ_2 of IF, the PC can be driven out to the external cache only in the next cycle. Three cache miss cycles were inserted at this point: *CM1* to drive the instruction address out to the external cache (normally a data address is potentially sent out), *CM2* for the external cache access, and *CM3* to write the instruction into the instruction cache and instruction register.

A three cycle penalty, with a miss rate of over 20%, degrades processor performance by over 60%. A cache miss penalty of two cycles, which reduces the loss to 40%, can be achieved by combining *CM2* and *CM3* into one cycle. The critical path now involves accessing the external cache memory, getting the data to the processor pads, and writing into the instruction cache. Since the data from the external cache arrives late in the cycle, this approach can easily affect the cycle time of the machine. Although it decreases the miss penalty, it might

[5]Note that our sub-block size of one word requires a valid bit for every word.

IF_{-1}	ϕ_1	Compute PC
	ϕ_2	Drive PC onto PCBus
IF	ϕ_1	Start tag access
	ϕ_2	Detect miss
CM_1	ϕ_1	
	ϕ_2	Drive PC out to external cache
CM_2	ϕ_1	
	ϕ_2	Instruction back from external cache
CM_3	ϕ_1	
	ϕ_2	Write instruction into c-cache and instruction register
RF	ϕ_1	Miss sequence completed
	ϕ_2	

Figure 10: Miss Timing for a C-cache.

increase the average cost of a fetch by increasing the cycle time of every instruction.

Clearly, if we require a minimum of two cycles to access the external cache and write the instruction into the instruction cache, the only way to reduce the miss penalty is to detect the miss sooner. If a miss can be detected before the end of ϕ_1 of IF, the PC can be driven out right away, eliminating one cache miss cycle. In the buffer scheme, driving the address to the cache array to fetch the valid bits causes the miss signal to appear late. The solution is to move the valid bits into the datapath along with the tags. Then, tag comparison and valid bit checking can be done a phase earlier.

There is one problem, however. To know which valid bit to fetch, we need to know which tag matched. Instead of accessing the valid bit after the tag comparison, we fetch all possible valid bits in parallel, one for each tag, along with the tag compare. The result of each tag compare is "AND-ed" with its corresponding valid bit. A cache miss occurs if none of these outputs is true. Figure 11 shows the miss timing for a buffer with the valid bits and tags stored in the datapath. The miss penalty is now reduced to two cycles.

An interesting and important side effect of moving the tags and the valid bits into the datapath is that the cache array becomes strictly a RAM array. With the valid bits in the cache array, the array would need to be customized to our specific cache configuration because the valid bits must be cleared when a new tag is written into that block. Now, the valid bit circuitry is independent of the RAM and also simpler to implement.

With a two cycle penalty, and a 20% miss rate, the performance

IF_{-1}	ϕ_1	Compute PC
	ϕ_2	Drive PC onto PCBus
IF	ϕ_1	Do tag compare; detect miss
	ϕ_2	Drive PC out to external cache
CM_1	ϕ_1	
	ϕ_2	Instruction back from external cache
CM_2	ϕ_1	
	ϕ_2	Write instruction into cache and instruction register
RF	ϕ_1	Miss sequence completed
	ϕ_2	

Figure 11: Miss Timing for a Buffer

loss is 40% which is still significant. We can reduce the miss penalty to one cycle by combining *CM1* and *CM2* into one cycle. As mentioned before, this extends the machine cycle time. An alternate solution is to use the extra cycle wisely to prefetch another word.

4.5.3 Prefetching

A side effect of the two cycle miss penalty in MIPS-X is that the timing allows fetching back not only the instruction that missed but also the next instruction (not the next sequential one, but the instruction to be executed next) during the extra cache miss cycle. This means that the worst miss rate for the cache is 50%, and the average miss rate is about half of what it would be without prefetching.

The method of prefetching an extra word can be explained with the aid of the cache miss timing in Figure 12. In the phase following miss detection for the current instruction (whose address is PC_{miss}), the address of the next instruction (PC_{next}) has already been calculated. This address would have been sent to the instruction cache in the normal sequence. While the external cache is being accessed, the next instruction address is set up on the address pads (in *CM1*); and while the missed instruction is written into the instruction cache during *CM2*, the external cache is accessed for the next instruction. Then, in the following phase (*RF*), when execution of the missed instruction is commenced, the next instruction is simultaneously written into the instruction cache and instruction register. Thus, the timing of the pipeline allows the prefetch to occur quite naturally.

Prefetching the extra word has a tremendous performance impact. For the MIPS-X hybrid buffer, the miss rate drops from 23.17% to 11.88%, or performance degradation drops from about 40% to 20%.

IF_{-1}	ϕ_1	Compute PC_{miss}
	ϕ_2	Drive PC onto PCBus
IF	ϕ_1	Do tag compare and detect miss; Compute PC_{next}
	ϕ_2	Drive PC_{miss} out to external cache
CM_1	ϕ_1	
	ϕ_2	$Instruction_{miss}$ back from external cache Drive PC_{next} out to external cache
CM_2	ϕ_1	
	ϕ_2	Write $instruction_{miss}$ into cache and instruction register $Instruction_{next}$ back from external cache
RF	ϕ_1	Miss sequence completed
	ϕ_2	Write $instruction_{next}$ into cache and instruction register

Figure 12: Fetching Back Two Words on a Miss

Note that this scheme has the effective performance impact of a one cycle miss penalty, but without the risk of increasing the machine cycle time. Implementation is also simple because fetching the second word fits in with the natural flow of the cache miss sequence. This shows that careful matching of the cache miss timing to the pipeline of the machine can give significant performance benefits.

Other prefetching schemes that exploit the available excess bandwidth were also considered. For example, in MIPS-X, when the processor is not fetching data, the I/O pins are free and instructions could be prefetched into the cache without affecting any other activity of the processor. However, these schemes could not be used in MIPS-X because the instruction cache does not have sufficient band width. MIPS-X uses 100% of the instruction cache bandwidth for fetches, preventing a prefetcher from using the cache. The instruction cache is only free during instruction cache misses. Thus, no prefetch scheme can do better without dramatically changing the cache organization.

4.6 Summary of Organization Choice

Table 2 summarizes the cache performance statistics for the various schemes assuming that two words are fetched back on a miss. Conventional cache organizations perform worse than buffers because of their high miss penalty. Note that the lowest miss rate does not yield the best performance. A hybrid buffer with four sets, associativity eight, and block size 64 bytes performs best with an access time of 1.24 cycles, and is used in the MIPS-X design.

Cache type	Implement-able	Num. Sets	Set size	Block size	Miss rate(%)	Miss penalty	T_{acc}
C-cache	YES	256	1	4	16.74	3	1.50
C-cache	NO	16	8	16	10.90	3	1.33
Buffer	YES	1	8	256	14.78	2	1.30
H. buffer	YES	4	8	64	11.85	2	1.24

Table 2: Summary of cache performance. Block size is in bytes; miss penalty and access time are in cycles.

5 Selection of Other Cache Parameters

5.1 Replacement

The replacement algorithm has traditionally been very important in cache design. Of the feasible schemes, least recently used replacement is considered to perform the best, although Smith and Goodman [21] show evidence that it might be inferior to random replacement for instruction cache design. After considering a number of replacement strategies, including LRU, RAND, FIFO, RING, and RING-M, we came to the conclusion that the replacement algorithm is not critical to the design for small caches.

LRU (least recently used) is where the least recently accessed block in any given set is replaced. In random replacement (RAND), a truly random choice of block to be replaced is made. FIFO is first in first out, where the block present the longest in any given set is replaced first. RING, is a pseudo-random replacement scheme where a ring counter with the same number of states as the set size is maintained. The counter points to the block in each set that must be replaced on a block miss or if an address tag does not match any of the cache tags. The counter is bumped one state after every block miss. RING-M is a modification of the above scheme.

Table 3 compares the relative performance of our hybrid buffer for the various replacement schemes. RAND, FIFO, RING, and RING-M have about equal performance. LRU is slightly better than the other schemes. For the MIPS-X design, we chose one of the RING schemes because of its simpler implementation. The RING-M scheme was used to solve a subtle problem with the double fetch on instruction cache misses. As stated before, two instructions, I_{miss} and I_{next}, are fetched on a cache miss. The problem arises when the first word (I_{miss}) hits in the tags, but is not valid yet, and the next instruction (I_{next}) misses

Replacement Scheme	LRU	RAND	FIFO	RING	RING-M
Miss Rate (%)	10.91	11.71	11.70	11.75	11.85

Table 3: Comparison of replacement schemes.

Flush Interval	5K	10K	20K	30K	40K	50K
Miss Rate (%)	14.00	12.98	12.43	12.28	12.12	12.09

Table 4: Effect of cache flushing on context switches.

in the tags.[6] If the ring counter points to the block that I_{miss} will occupy, then that block will be replaced by the tag corresponding to I_{next}, causing I_{miss} to have nowhere to go when it is received from the external cache. To avoid this state, we bump the counter if it points to a cache block corresponding to the most recent address tag.

5.2 Context Switch Mechanisms

Virtual caches, or caches addressed using the virtual address generated by the processor, have the advantage that virtual to physical translation is removed from the critical path. However, they have other multiprogramming related problems. For one, the integrity of a process address space is harder to maintain. A simple solution is to flush the cache on every process switch. The performance degradation due to cache flushes is not serious for small caches.

Table 4 shows the miss rate for a hybrid buffer flushed every Q instructions, where Q ranges from 5000 to 50000. A higher frequency of flushing is expected in time sharing workloads while batch jobs will be much lower. Flushing the entire cache is easily achieved in VLSI by providing a cache reset signal. However, a cache flush would require a special instruction. To avoid defining another instruction, we decided to use a simple software technique and trade off a little performance. The virtual address space is half system and half user. To flush the cache of user instructions, we cause the processor to jump to 32 specific system addresses making all the tags be in system space. This requires 32 extra instructions or 64 extra cycles[7] every cache flush. Even if cache flushing takes place, say, once every 20000 cycles, the miss rate would go up from 11.85% to 12.33%.

[6] I_{next} is not constrained to be in the same block as I_{miss}.
[7] Note that only 16 of the 32 instructions suffer cache misses.

5.3 Testability Features

We have included a number of features into the design to enhance the testability of the instruction cache and the processor. The instruction cache and the rest of the processor have separate power supplies making it possible to power the processor independent of the cache. An external signal (ICacheDisable) forces the processor to always cache miss on instruction fetches allowing the processor to run even if the cache is not completely functional.

The size of the instruction cache forced us to include a method to directly access the RAM. When an external signal called ICacheTest is asserted, the PC is forced to generate sequential addresses. These addresses will initially miss in the cache, and data will be loaded from the data pads. After filling up the cache, the processor can be reset and the entire cache read. Although this interface does not allow us to perform random reads and writes into the cache, it does let us directly test the basic functionality of the cache before we use it for supplying instructions to the processor.

6 Conclusions

Cache design has already been studied in great detail but only recently has it been feasible to implement caches on the same chip as a processor. We showed that for on-chip caches other considerations besides hit rate are important. These include the total usable area, the timing of cache accesses, the physical organization of the cache, and the aspect ratio of the resulting design. Minimizing the average instruction access time – a combination of both the miss rate and the miss penalty – is the key goal. We showed that given several physical organizations that satisfy the space and size constraints, the resulting miss rate can vary considerably, and the organization with the lowest miss rate does not necessarily result in the best performance.

We showed the importance of the tradeoffs between cache architecture and implementation by describing the design of a real on-chip cache for MIPS-X. Given an initial set of constraints for the physical dimensions and the cycle time of the desired cache, we used trace driven simulations to measure the performance of three basic cache configurations, varying several parameters such as set size, and block size. With these results, we computed the average instruction access times by taking into account the number of cycles needed to service a miss for each of the configurations, and made our choice based on the minimum average instruction access time. The result was a cache organization that is a hybrid between a conventional cache and an instruction buffer. This organization has a miss rate of roughly 12% for a set of large benchmarks, and results in an average instruction access time of 1.24 cycles. This penalty is roughly 3 times smaller than the penalty of our first cache organization, although the basic cache organization (cache size, block size, etc.) remained unchanged.

7 Acknowledgements

The MIPS-X research effort was supported by the Defense Advanced Project Research Agency under contract No. MDA903-83-C-0335. Paul Chow was partially supported by the Natural Sciences and Engineering Research Council of Canada.

We also gratefully acknowledge the contributions of Malcolm Wing, Karen Huyser, Scott McFarling, C. Y. Chu, Steve Richardson, Steve Tjiang, Richard Simoni, Don Stark, Glenn Gulak, and Steven Przybylski.

References

[1] Anant Agarwal, Mark Horowitz, and John Hennessy. *An Analytical Cache Model.* CSL 86-304, Stanford University, September 1986.

[2] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 119–127, June 1986.

[3] Donald Alpert. *Performance Tradeoffs for Microprocessor Cache Memories* . CSL 83-239, Stanford University, December 1983.

[4] Paul Chow. *MIPS-X Instruction Set and Programmer's Manual.* CSL 86-289, Stanford University, May 1986.

[5] *CRAY-1 Computer Systems, Hardware Reference Manual.* Cray Research, Inc., Chippewa Falls, WI, 1979.

[6] D. Alpert et al. 32-bit Processor Chip Integrates Major System Functions. *Electronics*, 56(14):113–119, July 1983.

[7] George S. Taylor et al. Evaluation of the SPUR Lisp Architecture. In *Proceedings of the 13th Annual Symposium on Computer Architecture*, pages 444–452, June 1986.

[8] J. L. Hennessy et al. Design of a High Performance VLSI Processor. In *Proceedings, Third Caltech Conf. VLSI*, pages 33-54, March 1983. California Institute of technology, Pasadena, CA.

[9] M. D. Hill et al. *SPUR: A VLSI Multiprocessor Workstation*. CSD 86-273, UC Berkeley, December 1985.

[10] M. Horowitz et al. A 32-Bit Microprocessor with 2K-Byte On-Chip Cache. In *IEEE International Solid-State Circuits Conference*, 1987.

[11] James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124-131, June 1983.

[12] J. L. Hennessy. VLSI Processor Architecture. *IEEE Transactions on Computers*, C-33(12), December 1984.

[13] Mark Hill and Alan Jay Smith. Experimental Evaluation of On-Chip Microprocessor Cache Memories. In *Proceedings of the 11th Annual Symposium on Computer Architecture*, pages 158-166, June 1984.

[14] M. Horowitz and P. Chow. The MIPS-X Microprocessor. In *Proc. WESCON 85*, 1985. IEEE, San Francisco.

[15] Doug MacGregor, Dave Mothersole, and Bill Moyer. The Motorola MC68020. *IEEE Micro*, 101-118, August 1984.

[16] First Look at Motorola's Latest 32-Bit Processor. Electronics, September 18, 1986.

[17] D. A. Patterson and C. H. Sequin. Design Considerations for Single-Chip Computers of the Future. *IEEE Transactions on Computers*, C-29(2):108-116, February 1980.

[18] G. Radin. The 801 Minicomputer. In *Proc. SIGARCH/SIGPLAN Symp. Architectural Support for Programming Languages and Operating Systems*, pages 39-47, March 1982. ACM, Palo Alto, CA.

[19] R. L. Sites. How to Use 1000 Registers. In *Proc., 1st Caltech Conf. VLSI*, January 1979. California Institute of Technology, Pasadena, CA.

[20] Alan Jay Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473-530, September 1982.

[21] James E. Smith and James R. Goodman. A Study of Instruction Cache Organizations and Replacement Policies. In *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 132-137, June 1983.

[22] W. D. Strecker. Cache Memories for PDP-11 Family of Computers. In *Proceedings of the 3rd Annual Symposium on Computer Architecture*, pages 155-158, January 1976.

VLSI IMPLEMENTATION OF A PROLOG PROCESSOR

Vason P. Srini, Jerric Tam, Tam Nguyen, Chien Chen
Allen Wei, Jim Testa, Yale N. Patt, and Alvin M. Despain

University of California
Berkeley, CA 94720

ABSTRACT

The current interest in the high performance execution of Prolog programs demands research in alternative architectures and efficient implementations. At Berkeley, we have developed a stack oriented architecture for Prolog and designed a VLSI chip using static CMOS technology with two layers of metal. The architecture is an adaptation of Dobry's TTL-PLM architecture based on Warren's Abstract Machine. It employs an environment stacking scheme. This paper describes the challenges encountered in designing the chip, strategies used to achieve the design goal of 100 ns cycle time (worst case), and tradeoffs employed to reduce the size of the chip to 12.5mm X 9 mm. The chip is a coprocessor chip and can be interfaced to standard buses such as VME and MULTIBUS II.

END

DATE
FILMED

7-87