Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors

Kenneth M. Wilson, Kunle Olukotun, and Mendel Rosenblum

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
{kwilson, kunle}@ogun.stanford.edu
mendel@cs.stanford.edu

Abstract

The memory bandwidth demands of modern microprocessors require the use of a multi-ported cache to achieve peak performance. However, multi-ported caches are costly to implement. In this paper we propose techniques for improving the bandwidth of a single cache port by using additional buffering in the processor, and by taking maximum advantage of a wider cache port. We evaluate these techniques using realistic applications that include the operating system. Our techniques using a single-ported cache achieve 91% of the performance of a dual-ported cache.

1. Introduction

It is well known that microprocessor performance is improving at a greater rate than the performance of the memory subsystem. One of the microprocessor design techniques responsible for this increase in microprocessor performance is dynamic superscalar computer architectures. Dynamic superscalar processors exploit instruction level parallelism and have large memory bandwidth demands. As the bandwidth of the memory subsystem has been increased to meet the demands of current processors, the single processor cache port has become one of the main performance bottlenecks of the processor. To remove this bandwidth bottleneck, current microprocessors have been built with dual-ported caches.

The two techniques for implementing multiple cache ports are to duplicate the cache and to interleave the cache. Ideally, for a processor with a dual-ported cache, each cache port would operate independently, doubling the number of potential cache accesses. However, these techniques may not attain the ideal performance of a multiple-port cache. An example of duplicating the data cache is implemented in the DEC Alpha 21164 [Gwen93]. DEC implements a dual-ported cache by maintaining identical copies of the data set in each cache. To keep both copies identical, all stores must be sent to both cache ports simultaneously. Since stores can be buffered and bypassed to allow loads to access the cache first, the hope is that the constraint of a single store will not significantly degrade performance. The cost of this improvement is a doubling of the die area of the primary data cache.

The second technique for implementing more than one cache port uses multiple independently-addressed banks to build an interleaved cache [Sohi91]. The MIPS R10000 [MIPS94] implements a two-way interleaved (banked) cache where all data references address either bank one or bank two. Splitting cache accesses into two banks allows the two banks to be addressed independently. Given the right cache access stream, this technique can double the bandwidth of the primary data cache. It is possible for bank conflicts to reduce the performance of an interleaved cache. The cost of implementing an interleaved cache is an additional multiplexer between the load/store unit and the cache ports.

Both the duplicate data cache and the interleaved data cache increase cache bandwidth by adding more cache ports. The problem is that both methods have significant costs or drawbacks. In this paper we propose and evaluate techniques for improving the primary cache bandwidth without extra cache ports. These techniques approach the performance improvement of multiple ports by using additional buffering in the processor, and by taking maximum advantage of the bandwidth of a single cache port. We consider four methods that build upon one another for increasing the cache port bandwidth: *load all* (LA), *load all* wide (LAW), *keep tags* (KT), and *line buffer* (LB). Load all increases cache bandwidth by satisfying as many outstanding loads as possible when data is returned from the cache. Dynamic superscalar processors lend themselves well to the implementation of load all, because the load/store unit can be implemented as a set of buffers that snoop on the data returning from the cache. Load all wide builds upon load all by widening the single cache port to increase cache bandwidth. Keep tags improves upon load all by preventing an access to the cache that will reference a cache line that has an outstanding miss. Preventing multiple misses to the same cache line frees the cache port for accesses that may hit in the cache. Of course, this improvement

requires the use of a non-blocking or hit-under-miss cache [Fark94]. Line buffer keeps the cache data inside the load/store unit. This allows accesses to recently referenced data to be satisfied from the line buffer instead of from the cache. Essentially, the line buffer acts as a level-zero data cache.

In this paper we show that the four methods described above significantly improve the performance of a dynamic superscalar processor with a non-blocking cache and one cache port. The rest of this paper is organized as follows: The computer architecture used in this investigation is described in section 2, and the experimental methodology used to simulate the architecture described in section 2 is discussed in section 3. Section 4 contains the results characterizing the design space by simulating the interactions between different cache bandwidth optimizations. Section 5 presents the conclusions drawn from the experimental results and the preceding discussion.

2. Architectural Assumptions

This section describes the computer architecture that is used to simulate the results presented in this paper. The details of our dynamic superscalar processor and memory model are presented below, followed by descriptions of possible implementations of the four proposed hardware enhancements: load all, load all wide, keep tags, and line buffer.

2.1. Processor and Memory Subsystem

The processor used for this study is a four issue dynamic superscalar processor with R10000 instruction latencies and a 200 Mhz clock rate [Gwen94, MIPS94]. Unlike the R10000, our processor model has a 64 entry reorder buffer, and a 32 entry load/store buffer. To focus our attention on the performance of the memory subsystem, there are no restrictions on the type of instructions that can be issued each cycle.

The memory subsystem consists of a separate primary instruction and data cache, a unified secondary cache, and main memory. The primary instruction cache is perfect and responds in a single cycle. The primary data cache is a two-way-set-associative cache with 32 byte lines and a single cycle access time. The data cache is lock-up-free, but the cache size is not fixed to allow us to investigate the performance effects of various cache sizes. The second level cache is a 4 Mbyte two-way-set-associative cache with 64 byte lines and a ten cycle (50ns) access time. The four miss status handling registers (MSHR) [Fark94] implementing the lock-up-free cache are located in the primary data cache to support misses for up to four data cache lines. Main memory has a sixty cycle (300ns) access time.

2.2 Load Instruction Execution

To understand the implementation of the four proposed hardware enhancements, we describe the execution path for a load instruction within the processor's load/store unit. The first step in the execution of a load instruction is to generate the effective address. After the effective address has been calculated, dynamic memory disambiguation is performed in the store forwarding unit in parallel with loading the cache access buffer, as shown in Figure 1. All loads in the cache access buffer that are not dependent upon a store will access the cache in FIFO order. When the load is placed into the

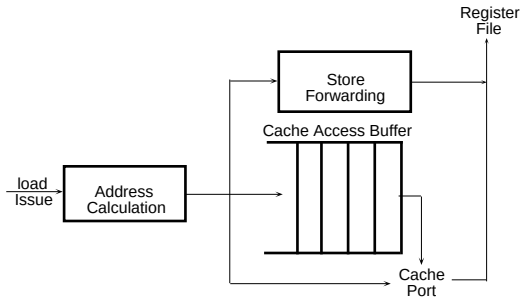

Figure 1: The load/store unit datapath.

cache access buffer, and the cache port is not in use, the load is sent directly to the cache while the cache access buffer is being loaded; this avoids a one cycle delay in the accessing of the data cache.

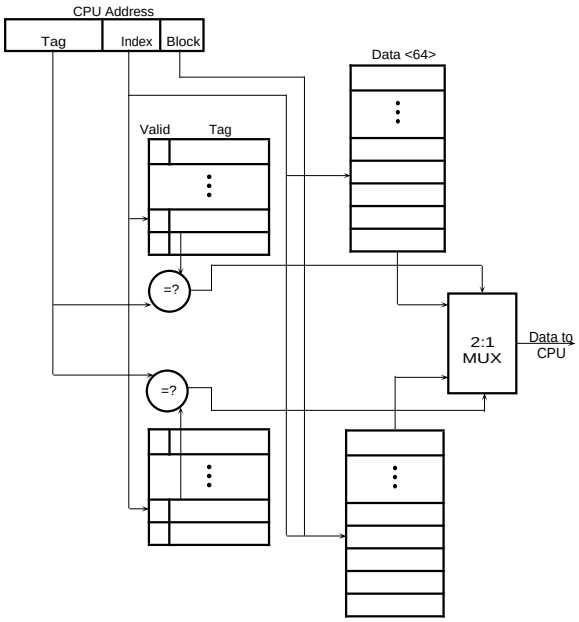

Figure 2: A two-way-set-associative cache

2.3. Load All

The load all enhancement increases cache bandwidth by allowing multiple loads to be satisfied by returning cache data in parallel. Satisfying load instructions in parallel does not affect program correctness because dynamic memory disambiguation has already removed all loads that depend upon a store instruction that has not yet been completed. Dynamic memory disambiguation is performed in the store forwarding unit.

To satisfy a load instruction from the returning cache data, each cache access buffer entry contains a comparator. The comparator is used to compare the tag of the address of the cache access with the tag of the buffered load instruction's address in parallel with the cache access. If the tags are equal, then the cache access buffer can be filled with the returning data. In this way, one cache access can satisfy multiple load instructions without increasing the cache access time.

2.4. Load All Wide

The load all enhancement can be further improved by widening the processor's cache port, increasing the number of loads that can be satisfied with the returning data in parallel. To explain how the widening of the cache port is implemented, the block diagram for a two-way-set-associative cache is shown in Figure 2. In this figure, the output of the cache is the cache block that the load references, whereas the load all enhancement can handle as much as an entire cache line returning from the cache. Figure 3 shows how Figure 2 would have to be modified so that the cache returns an entire line. The decoding of the data portion of the cache is reduced, since the output of the data section of the cache is now an entire line, instead of one cache block. Note that increasing the size of the data returning from the cache will increase the number of sense amps needed and therefore change the aspect ratio of the cache.

To make use of an entire cache line, each cache access buffer entry must now contain a multiplexer as well as a comparator as shown in Figure 4. If the comparator detects that the tags are equal, then the multiplexer is used to select the correct data block from the returning cache line. In this way, one cache access can satisfy multiple load instructions while only increasing the cache access time by the delay of one multiplexer less the reduction in decoder time for the cache to decode a larger data chunk.

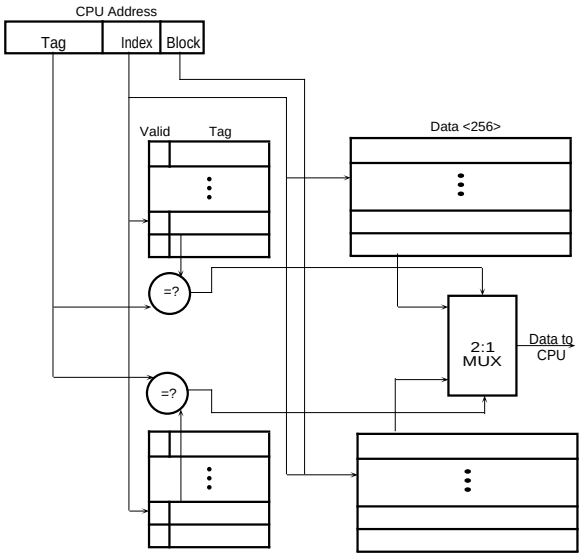

Figure 3: A two-way-set-associative cache for load all wide.

2.5. Keep Tags

The *keep tags* enhancement could be combined with the load all or load all wide enhancement to reduce pressure of on cache port by removing cache accesses that are sure to miss. These cache accesses are removed by adding a small tag buffer that contains the tags of all outstanding data cache misses as shown in Figure 5. As each load instruction is entered into the cache access buffer, the tags stored in the tag buffer are checked against the address tag of the returning load instruction. If there is a match, a bit is set in the cache access buffer to prevent cache access by this load instruction.

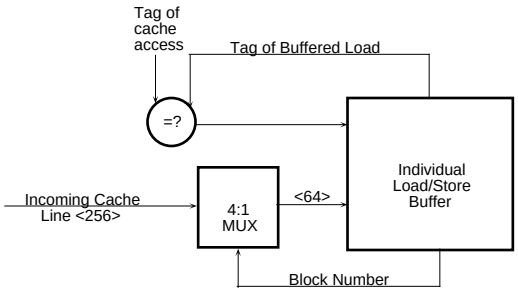

Figure 4: The load all wide hardware enhancement.

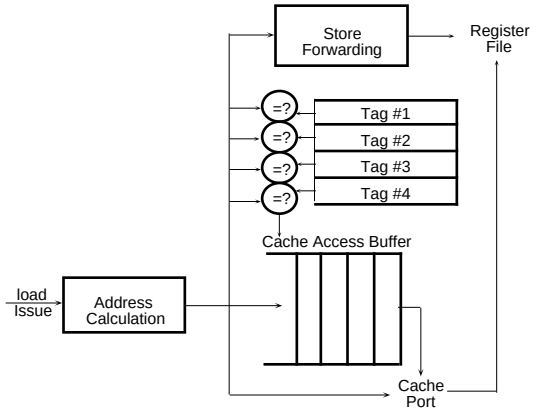

Figure 5: The keep tags hardware enhancement.

Since the tag buffer logic is totally separate from the logic of the cache access buffer, it should be possible to perform the tag compare in parallel with the loading of the cache access buffer. If the cache access buffer contains no load waiting for cache access, the returning load will still bypass to the cache as explained in section 2.3. Even if the load is to a cache line with an outstanding miss, bypassing to the cache will not degrade the performance of keep tags, because the cache bandwidth used is not needed by any other loads.

The maximum number of tag buffer entries needed to hold the tags of outstanding data cache misses is the number of MSHRs in the data cache (four in this case), because the processor's lock-up-free cache can contain at most one outstanding cache line miss per MSHR. Moreover, Farkas and Jouppi's [Fark94] results suggest that only a small tag buffer would be needed as there are unlikely to be very many data cache misses outstanding to multiple cache lines at one time. Note that if the MSHRs are implicitly or explicitly addressed and located in the load/store unit instead of in the data cache, then the keep tags improvement is already implemented inside the MSHRs.

2.6. Line Buffer

When data is returned from the cache, load all satisfies all outstanding loads to that data, but does not buffer the data. The line buffer improves upon the load all enhancement by retaining recently fetched data in a buffer. This buffer is then used to satisfy loads that hit in the buffer. Think of the line buffer as a small multi-ported ful-

ly-associative level-zero cache with a FIFO replacement policy. To implement the line buffer, the address tag and data of the returning cache access would be stored in the line buffer, replacing the oldest entry, as shown in Figure 6.

While a load is being placed in the cache access buffer, the tags stored in the line buffer are checked to see if the line buffer contains the data requested by the load. If the requested data is found in the line buffer, then the data is returned from the line buffer instead of accessing the cache. If the cache access buffer contains no loads ready to access the cache, the returning load will still access the cache even if the line buffer contains the data, because these two operations are performed in parallel. Accessing the cache in parallel when the requested data is already in the line buffer will not cause a problem, because the returning data from the new cache access will be entered in the line buffer as usual.

Note that the line buffer only needs to be checked as the load instruction is placed into the cache access buffer, because any load entries waiting in the cache access buffer will be completed by load all. The line buffer is accessed by load instructions as they issue; to be most effective, the line buffer must have as many ports as the number of loads that can be issued in a single cycle. Our four-issue processor has no issue restrictions so the line buffer in this model must have four ports. Multi-porting the line buffer is much easier than multi-porting the cache due to the small size of the line buffer.

When a store instruction addresses a cache line in the line buffer, the line must be updated or invalidated. If this processor were used as part of a multiprocessor, a cache line in the line buffer would have to be invalidated when the corresponding line was invalidated in the primary data cache to keep the cache coherent. The invalidation could be performed by snooping primary data cache invalidations so that the primary data cache and the line buffer are invalidated in parallel. Note that both keep tags and line buffer require load all or load all wide; however, keep tags and line buffer could be combined to implement a fifth option.

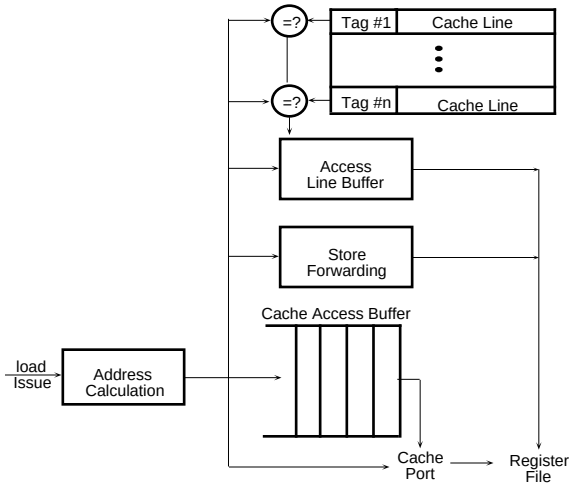

Figure 6: Line buffer hardware enhancement.

3. Simulation Methodology

This section describes the SimOS environment and how we use it to simulate the data cache architectures described in the previous section. We also discuss the simulation methods and benchmarks used to produce the results presented in this study.

3.1. SimOS Environment

SimOS [Rose95b] is a complete machine simulation environment containing models of all the hardware present on modern computer systems including the CPU, memory subsystems, and I/O devices. It can simulate the hardware with enough speed and detail to boot and run a commercial Unix-based operating system, Silicon Graphics IRIX version 5.3, and arbitrary application programs that run on IRIX. Although a full description of SimOS is beyond the scope of this paper, we provide a brief description of the features of SimOS most relevant to this study.

The use of a commercial operating system in SimOS provides great flexibility in the programs and workloads used to evaluate the extensions proposed in this paper. From the large base of applications that run on IRIX we chose programs from the standard benchmark suite SPEC95 as well as large commercial applications including relational database systems and CAD packages. Section 3.2 describes these workloads in detail.

A second useful feature of SimOS is a high speed simulation mode that employs on-the-fly binary translation [Witc96] to allow workloads to run only a factor a 10 times slower than the workloads would on the machine running the simulator. We use this high speed mode to boot IRIX on the machine simulator and to start and configure the workload to be studied. For the complex workloads, this boot and setup phase required tens of billions of instructions, making the fast simulator a necessity. With the fast mode we can setup even the most complex workloads and ensure the workload is past any startup transient conditions.

A third feature of SimOS is the checkpointing mechanism that allows the entire state of the simulation machine to be written to a set of files. These checkpoints included the state of all registers, memory, and I/O devices of the simulated machine. This checkpoint can then be restored into SimOS to start simulating from the point in the execution at which the checkpoint was made. Each workload we studied had a checkpoint from which all simulation studies started, avoiding the time-consuming simulation of the IRIX boot and workload start-up for every simulation run.

The actual studies of cache bandwidth enhancement were performed using the MXS [Benn95] CPU simulator. MXS is a detailed CPU simulator that simulates the machine model described in section two. This CPU simulator models a dynamic superscalar processor that supports multiple instruction issue, out-of-order execution, hardware branch prediction, and lock-up-free caches. The simulator's operation is parameterized and can be configured to accurately model a variety of machines.

Due to the slow simulation speed of MXS (over 10,000 times slowdown) and the long running times of the workloads, it is infeasible to run the entire workloads under MXS. For long running workloads we employed the sampling feature of SimOS that allows

switching between the slow but detailed MXS simulator and Mipsy, a much faster simulator that contains a simple pipeline model. Since Mipsy and MXS share the same cache simulation, it was possible to switch between them with little overhead. Sampling allowed the MXS simulator to "see" a much larger portion of the workload with significantly less simulation time. Details of the sampling used varied depending on the workload characteristic and are described in the following section.

3.2. Benchmarks

The performance of nine realistic benchmarks are used to evaluate the hardware enhancements that are proposed in section two. Table 1 shows that the nine benchmarks are made up of three integer SPEC95 benchmarks, three floating point SPEC95 benchmarks, and three multiprogramming applications. The newer SPEC95 benchmarks were chosen over SPEC92 benchmarks because they stress the memory system much more than the SPEC92 benchmarks. The three SimOS benchmarks were chosen because they represent realistic multiprogramming workloads from the areas of programming, database transaction processing, and engineering simulation. When detailed analysis of the benchmarks are required, one benchmark from each of the three categories will be chosen to present the results.

SPEC95 integer benchmarks	
gcc	Builds SPARC code
li	LISP interpreter
compress	Compresses and decompresses file in memory
SPEC95 floating point benchmarks	
tomcatv	Mesh-generation program
su2cor	Quantum physics; Monte Carlo simulation
apsi	Solves problems regarding temperature, wind, velocity, and distribution of pollutants
SimOS multiprogramming benchmarks	
pmake	Two compilation processes for 17 files
database	Sybase SQL server using bank/customer transaction processing modeled after the TPC-B transaction processing benchmark [Gray93]
VCS	Simulates the FLASH MAGIC chip [Kusk94] using the Chronologics VCS simulator

Table 1: The nine benchmarks.

During simulation, we dynamically switch between the CPU simulators MXS and Mipsy. For the SPEC95 benchmarks, a checkpoint was taken 120 seconds into the execution of the simulated benchmarks. This represents about ten percent of the run-time of the benchmarks, and is performed to remove any effects from initialization code contained in the benchmarks. In addition, Mipsy is run for 100,000 cycles before MXS starts collecting statistics to avoid most of the compulsory misses. After the checkpoint is restarted, Mipsy and MXS run with a ratio of 100,000 cycles to 10,000 cycles respectively, and the simulation is stopped when MXS has simulated one hundred million user instructions. Since Mipsy and MXS share the same caches, running 10,000 instructions in MXS should be more than enough to fully exercise the internal structures of the CPU. The dynamic switching between Mipsy and MXS results in data being collected for 1/11th of the benchmark execution time. Since Mipsy is so much faster than MXS this sampling allows for the simulation of a larger portion of the benchmark with only a small increase in simulation time.

	Kernel	User	Idle	Load	Store
gcc	10.0	90.0	0.0	28.1	12.2
tomcatv	0.4	99.6	0.0	26.9	8.5
database	18.4	17.0	64.6	24.8	13.6
VCS	9.9	90.1	0.0	25.7	15.1
pmake	8.9	86.0	5.1	25.8	11.9
li	0.2	99.8	0.0	33.2	13.0
compress	8.4	91.6	0.0	34.5	8.0
apsi	2.2	97.8	0.0	40.0	11.7
su2cor	0.5	99.5	0.0	28.0	6.3

Table 2: Execution time percentages plus percentages of loads and stores in the instruction stream.

The only exceptions to this sampling procedure are the floating point SPEC95 benchmarks. SimOS does not currently support dynamic switching for floating point benchmarks. Instead, the floating point benchmarks are run continuously for 100 million user instructions in MXS. The simulating of a smaller portion of the floating point benchmarks should not cause any problem because floating point benchmarks tend to contain small inner loops that are executed repeatedly with stable memory reference patterns.

Even though we are simulating for 100 Million user instructions, each benchmark runs for a different amount of time based upon its instructions per cycle (IPC) and the time spent in user mode, kernel mode, and idle. The breakdown of execution time spent in Kernel, User, and Idle for each benchmark is shown in Table 2. Idle is a result of waiting for I/O, and applications like database spend a large portion of their execution in this mode. Since the processor is spinning in a tight loop in idle mode, the IPC for this mode could skew the results for benchmarks with significant idle time. For this reason, the IPC of the benchmarks during idle time are not factored into our performance measurements.

It is interesting to note that the floating point benchmarks almost never execute in kernel mode, while the integer benchmarks spend up to ten percent of their execution time in kernel mode. If kernel mode execution was not simulated for the gcc benchmark, ten percent of its execution time would not be recorded, possibly causing an incorrect performance to be reported for gcc. In addition to improving the accuracy of simulation results for integer benchmarks, the simulation of kernel mode, as well as user mode, allows the simulation of multiprogramming benchmarks which spend 9% to 18% of their time in kernel mode.

4. Results

To characterize the performance of multi-ported caches and the per-

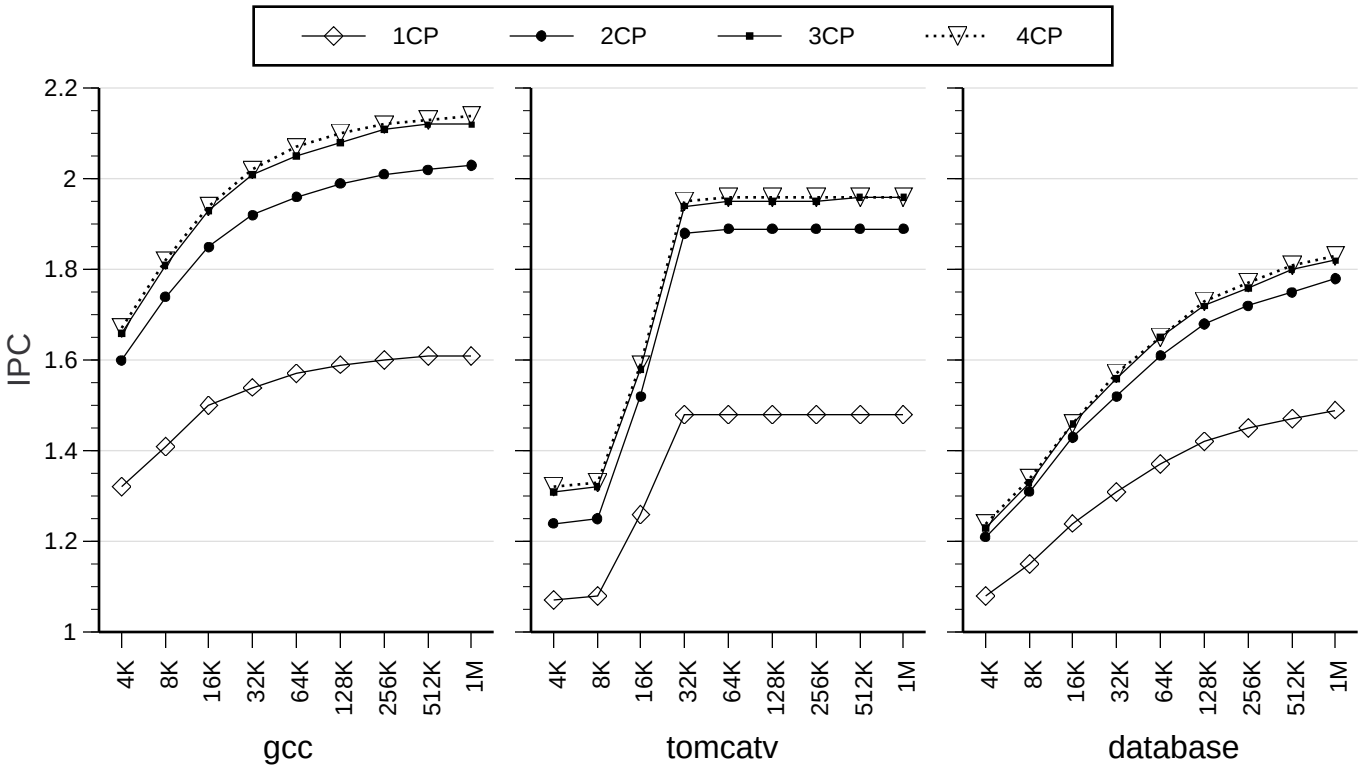

Figure 7: Performance of gcc, tomcatv, and database for one to four ideal cache ports.
Primary data cache size is varied from 4 Kbytes to 1 Mbyte.

formance benefits of the proposed hardware enhancements we simulate these organizations with cache sizes varying from 4 Kbytes to 1 Mbyte. The results of these simulations will be showcased with one benchmark from each of the three groups. Gcc, tomcatv, and database will represent the integer SPEC95 benchmarks, the floating point SPEC95 benchmarks, and the multiprogramming benchmarks respectively. At the end of this paper, the simulation results for all nine benchmarks with a 32 KByte data cache will be presented to summarize the performance of the proposed hardware enhancements. The simulation results provide us with the answer to the following key questions. First, what is the effect of increasing the number of cache ports on the performance of the benchmarks? Second, how do the four enhancements improve performance on the benchmark applications? Lastly, how does processor performance change as the number of entries in the line buffer is increased?

4.1. Multiple Cache Ports are Important

We have stated that the performance of the benchmarks used in this study will increase with the number of cache ports. Figure 7 shows the effect on performance of increasing the number of ideal cache ports for: database, gcc, and tomcatv benchmarks. Notice that for a 32 KByte primary data cache size, increasing the number of ideal cache ports from one to two shows a 25% improvement in performance for gcc, a 27% improvement for tomcatv, and a 16% improvement for database. The average performance improvements for an increase of two to three ideal cache ports is under 4%, and an increase from three to four ideal cache ports improves performance less than one percent.

Figure 7 clearly shows that a dynamic superscalar processor with two ideal cache ports produces a large performance improvement over the same processor with one cache port. The increase in performance is due to the second cache port doubling the bandwidth between the load/store unit and the cache. The purpose of the enhancements proposed in this paper is to increase the bandwidth of a single cache port to achieve performance improvements.

4.2. Load All and Load All Wide

To evaluate the increase in processor performance due to the increase of effective cache port bandwidth by the hardware enhancements load all and load all wide, simulation results for gcc, tomcatv, and database are shown in Figure 8. Figure 8 contains simulations for one cache port (1CP), load all (LA), load all wide (LAW), and two cache ports (2CP) for cache sizes of 4 KBytes to 1 Mbyte. Keep tags wide (KTW) is also contained in this figure for discussion in the next subsection.

Figure 8 shows that the increase in effective cache port bandwidth due to load all does increase processor performance and is fairly independent of cache size. The load all technique improves cache bandwidth by satisfying all loads waiting for the returning cache data, independent of whether the data is returned from a hit or a miss. This means that the performance gains due to load all are independent of miss rate, and therefore independent of cache size.

For the three benchmarks shown in Figure 8, load all increased processor performance by seven percent over a single unenhanced cache port. When the width of the cache port is increased to a full cache line of sixty-four bytes, the resulting processor performance

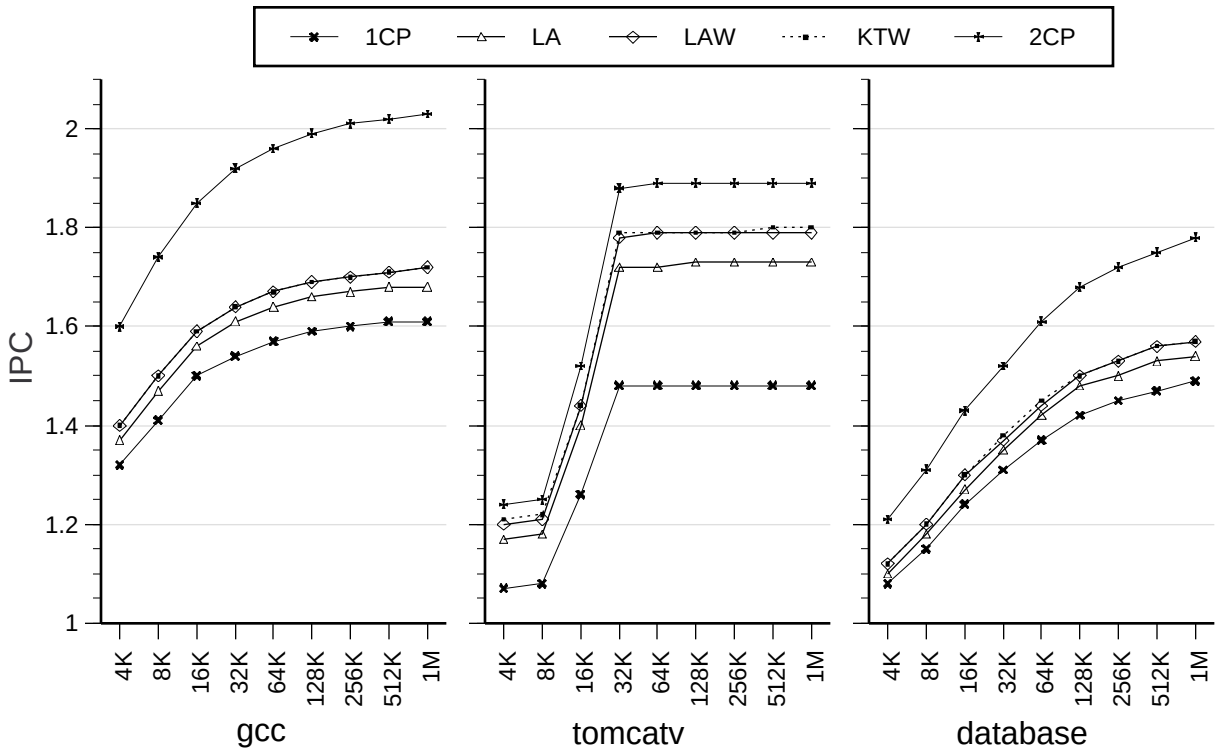

Figure 8: Load all and keep tags performance for gcc, tomcatv, and database.

is ten percent greater than the same processor with an unenhanced cache port. These results show that on average, the load all improvement achieves 40% of the performance increase of a dual-ported cache.

To give insight into how load all wide achieves an increase in processor performance, we present where within the load/store unit a load instruction is satisfied. Tables 3, 4, and 5 contain these statistics for the benchmarks gcc, tomcatv, and database respectively. The IPC and speedup for each configuration are also shown for completeness. As expected, one hundred percent of all load instructions are satisfied by accessing the primary data cache through a single-ported cache with no performance enhancements. With load all, a large fraction of loads (12-34%) are satisfied without accessing the data cache port, freeing up the cache port for accesses that might have had to wait multiple CPU cycles to access the data cache.

4.3. Keep Tags

Figure 8 also includes a curve showing the processor performance of keep tags wide (KTW) for the benchmarks gcc, tomcatv, and database. Keep tags wide does not perform well; showing only a very small performance improvement. The performance of keep tags wide as shown by this figure is representative of the performance seen by keep tags (KT), line buffer with keep tags (L8KT), and line buffer wide with keep tags (L8KTW). The poor performance of all of the enhancements using keep tags is expected for the integer benchmarks because they rarely have more than one miss outstanding [Fark94]. However, this was not expected for the three floating point benchmarks with more than one miss outstanding for 25% of the time for tomcatv, 34% for su2cor, but only 10% for apsi.

The only benchmarks with enough outstanding cache misses to have the potential for keep tags to improve processor performance are tomcatv and su2cor. The cache access patterns for tomcatv and su2cor are shown in Table 6 along with the percentage of cycles spent with no misses, one miss, and greater than one miss. From Table 6, keep tags reduces the number of misses by only allowing one outstanding miss per cache line. The sharp reduction in the percentage of two or more outstanding misses reveals that there are rarely any outstanding misses to more than one cache line. This means that there is little opportunity for keep tags to produce an improvement in performance.

Tomcatv's performance does not improve with keep tags, while su2cor's performance increases by two percent. The obvious reason for this is that su2cor has more outstanding misses than tomcatv. The real reason is that when tomcatv misses, if there exists a second miss to another cache line, the second miss tends to be the next cache access, and is therefore already accessing the cache as soon as possible.

4.4. Line Buffer

Figure 9 shows how line buffer and line buffer wide affect processor performance for gcc, tomcatv, and database. The increase in processor performance due to line buffer is consistent over a wide range of cache sizes for gcc and database, but not for tomcatv. The processor performance for tomcatv with line buffer outperforms tomcatv run on a dual-ported cache for small cache sizes. This happens because tomcatv's data set does not fit into the first level data cache for cache sizes under 32 KBytes. With cache sizes under 32 KBytes, the line buffer acts as an additional small multi-ported fully-associative level-zero cache.

	Load All	Line Buffer	Port 1	Port 2	Port 3	Port 4	IPC	Speedup
1CP			100.0%				1.54	0%
LAW	13.1%		86.9%				1.64	6%
L8W	6.6%	33.6%	59.8%				1.72	12%
2CP			81.9%	18.1%			1.92	25%
3CP			77.3%	16.1%	6.7%		2.01	31%
4CP			75.3%	15.2%	6.6%	3.0%	2.02	31%

Table 3: Breakdown of where the load was satisfied for gcc.

	Load All	Line Buffer	Port 1	Port 2	Port 3	Port 4	IPC	Speedup
1CP			100.0%				1.48	0%
LAW	33.7%		66.3%				1.78	20%
L8W	17.2%	31.4%	51.4%				1.83	24%
2CP			71.6%	28.4%			1.88	27%
3CP			72.9%	24.7%	2.4%		1.94	31%
4CP			70.2%	26.6%	2.1%	1.1%	1.95	32%

Table 4: Breakdown of where the load was satisfied for tomcatv.

	Load All	Line Buffer	Port 1	Port 2	Port 3	Port 4	IPC	Speedup
1CP			100.0%				1.31	0%
LAW	11.8%		88.2%				1.37	5%
L8W	7.1%	21.3%	71.6%				1.40	7%
2CP			82.8%	17.2%			1.52	16%
3CP			78.4%	16.1%	5.5%		1.56	19%
4CP			76.7%	15.1%	6.0%	2.2%	1.57	20%

Table 5: Breakdown of where the load was satisfied for database.
Note that for Tables 4, 5, and 6 a 32 Kbyte primary data cache is used. The first column of the table represents which cache port configuration was used. Both load all wide and line buffer wide are using a single cache port with a width of 32 bytes.

	Enhancement	No Misses	1 Miss	>1 Miss
su2cor	1CP	72	1	26
	LAW	67	2	31
	KTW	67	24	9
	L8W	66	2	32
	L8KTW	66	24	10
	2CP	66	1	34
tomcatv	1CP	81	2	17
	LAW	78	3	19
	KTW	77	19	4
	L8W	76	3	21
	L8KTW	76	20	4
	2CP	74	1	25

Table 6: Percentage of misses outstanding in a given cycle for the benchmarks su2cor and tomcatv.

From Tables 3, 4, and 5 we found that 12-34% of all loads are handled without accessing the cache port for load all wide. When line buffer wide is added, not only are load instructions waiting for data satisfied by load all, but future loads to the same cache line will be satisfied by the line buffer until the cache line is replaced from the buffer. This increases the total percentage of loads satisfied without accessing the cache to 28-49%; thereby increasing processor performance by eleven percent for line buffer, and fifteen percent for line buffer wide as compared to the same processor with an unenhanced single-ported cache. Note that to realize this performance improvement, only eight entries were use for both line buffer (64 bytes) and line buffer wide (256 bytes).

4.5. Changes in Buffer Size

Up to this point, all of the simulations for line buffer have focused on an eight entry line buffer. Figure 10 shows how performance increases with the number of entries in the line buffer for the benchmarks gcc, tomcatv, and database with a 32 Kbyte primary data cache. Note that for database, tomcatv, and gcc, line buffer wide with eight entries is past the knee of the performance curve. For line buffer, processor performance is still linearly increasing be-

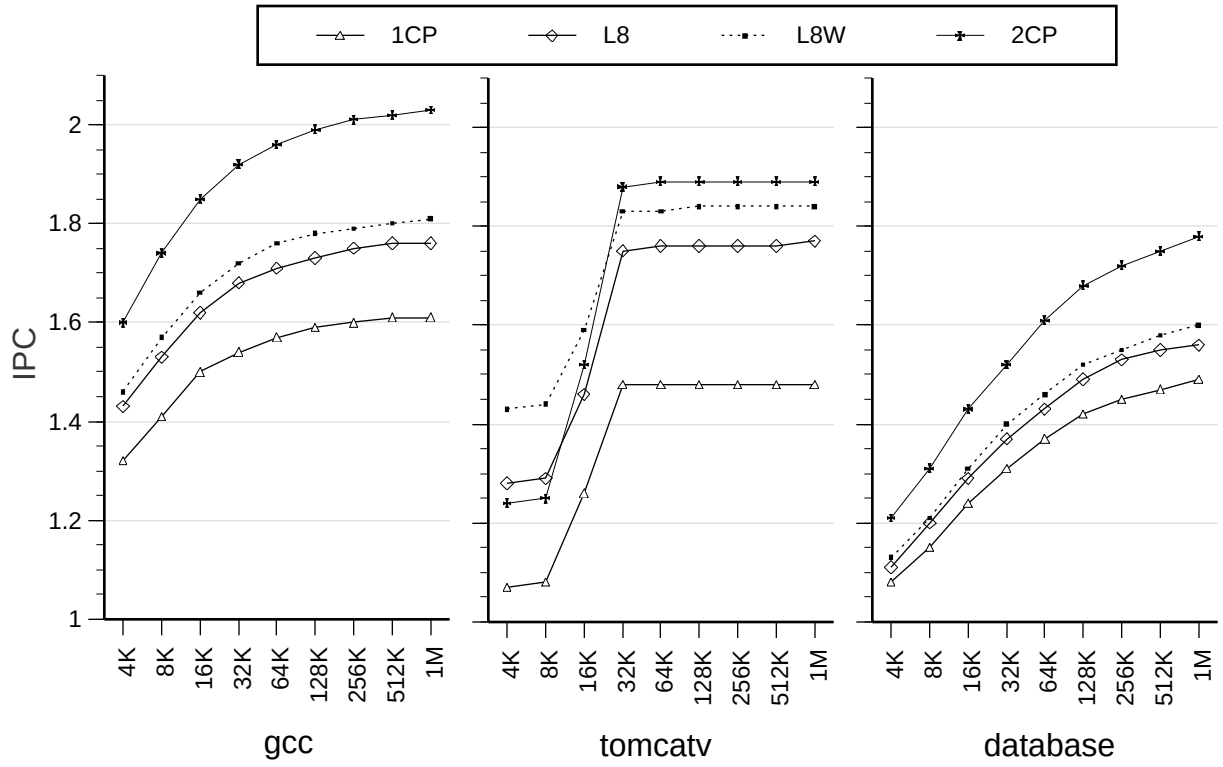

Figure 9: Line buffer performance.

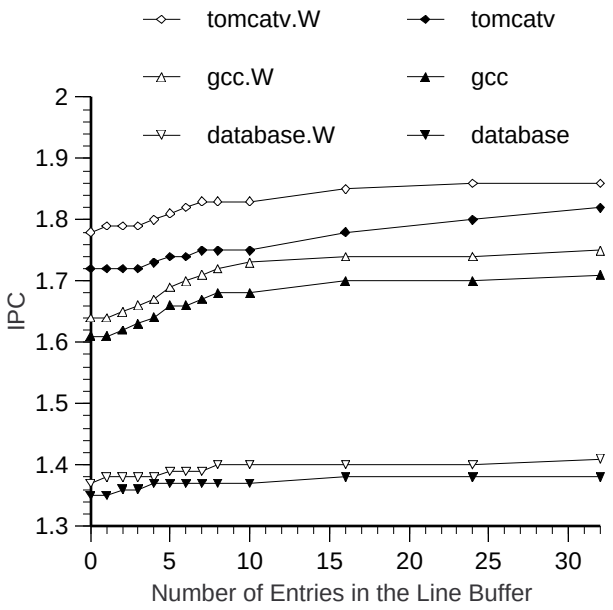

Figure 10: Impact of changing line buffer size.

tween eight entries and thirty-two entries.

4.6. Performance Summary for All Nine Benchmarks

We have presented detailed performance data on the enhancements gcc, tomcatv, and database. Figure 11 summarizes the results for all nine benchmarks with a 32 KByte primary data cache. The bar graph in Figure 11 includes processor performance for two ideal cache ports (2CP) as opposed to one cache port with: no improvements (1CP), load all (LA), load all wide (LAW), line buffer with eight entries (L8), and line buffer wide with eight entries (L8W). Since section 4.5 shows that processor performance is still increasing when line buffer is increased to thirty-two entries, the results for line buffer with thirty-two entries (L32) has also been included. Keep tags was not included in Figure 10 because keep tags and keep tags wide produced less than a one percent processor performance gain when applied to load all and line buffer.

The results presented in Figure 11 show that a processor with a single-ported cache will gain 8% in performance when load all is added, and 10% when using load all wide. The least gain in performance with load all was the benchmark compress with only a one percent gain, while the best increase in processor performance was achieved by tomcatv. The average performance improvement with two ideal cache ports is 25%. These results show that on average, the load all wide improvement achieves 40% of the performance increase of a dual-ported cache.

The addition of an eight entry line buffer (64 bytes) produces an average processor performance improvement of 11% over a processor with an unenhanced single-ported cache. When line buffer wide is used, the performance improvement increases to 14%. For line buffer, the database benchmark shows the least improvement in performance, while tomcatv once again achieves the most benefit from line buffer and line buffer wide.

The performance of line buffer (cache port width of eight bytes) can be increased to equal the performance of line buffer wide (cache port width of 32 bytes) by increasing the number of entries in the

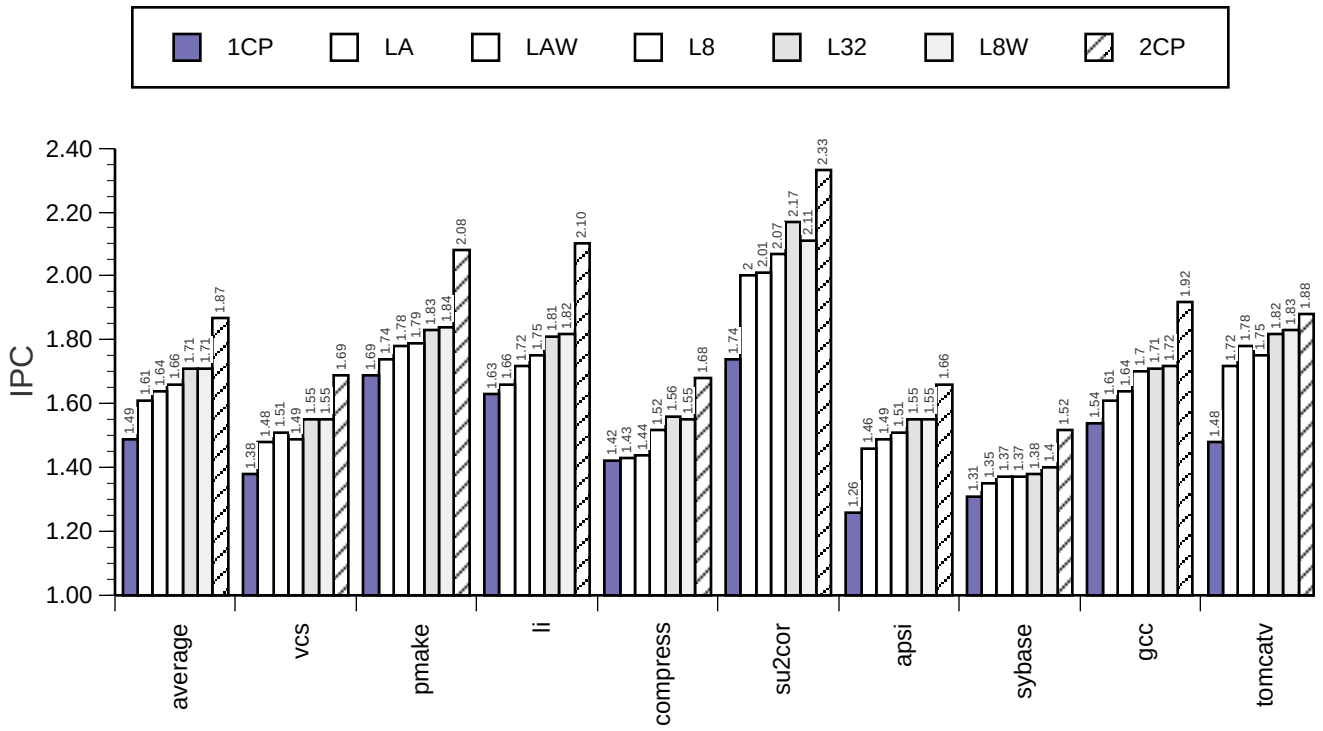

Figure 11: Benchmark performance with a 32 KByte data cache.

line buffer to thirty-two. Increasing the number of entries to thirty-two makes the two buffers equal in size. The extra associativity of the 32 by 8 byte buffer makes up for the extra prefetching available in an 8 by 32 byte buffer.

5. Conclusion

In this study we have proposed the load all, keep tags, and line buffer techniques for improving primary data cache bandwidth. We have evaluated these techniques with simulations that include operating system references for nine realistic benchmarks with primary data cache sizes varying from 4 Kbytes to 1 Mbytes. Our results show that load all wide can increase the performance of a four issue superscalar dynamic processor with only one cache port by an average of 10%, and the addition of a wide eight entry line buffer improves processor performance by 14%. Furthermore, the line buffer's performance is equivalent to 91% of the performance of an ideal dual-ported cache, and in the case of tomcatv, the line buffer technique performs almost as well as the dual-ported cache. A line buffer with as few as eight entries can capture a large percentage of the performance improvement possible from the line buffer enhancement. The only disappointment is the keep tags hardware enhancement. Keeping track of the tags of outstanding misses to the data cache produces at most a two percent increase in processor performance, and on average improves processor performance by less than one percent.

Acknowledgments

We would like to thank Steve Herrod, Edouard Bugnion, and Robert Bosch for their help with SimOS, and the reviewers for their insightful comments. This work was supported by ARPA contract DABT63-94-C-0054.

References

[Aspr93] Tom Asprey, Gregory S. Averill, Eric DeLano, Russ Mason, Bill Weiner, and Jeff Yetter, "Performance Features of the PA7100 Microprocessor", IEEE Micro, June 1993, pp. 22-35.

[Benn95] James Bennett and Mike Flynn, "Performance Factors for Superscalar Processors", Technical Report CSL-TR-95-661, Computer Systems Laboratory, Stanford University, Feb. 1995.

[Chap91] Terry I. Chappell, Barbara A. Chappell, Stanley E. Schuster, James W. Allen, Stephen P. Klepner, Rajiv V. Joshi, and Robert L. Franch, "A 2-ns Cycle, 3.8-ns Access 512-kb CMOS ECL SRAM with a Fully Pipelined Architecture", IEEE Journal of Solid-State Circuits, Vol. 26, No. 11, November 1991, pp. 1577-1585.

[Chen92] Tien-Fu Chen and Jean-Loup Baer, "Reducing Memory Latency via Non-blocking and Prefetching Caches", ASPLOS-V, Boston, Massachusetts, October 12-15, 1992.

[Chen94] Chung-Ho Chen and Arun K. Somani, "A Unified Architectural Tradeoff Methodology", ISCA-21, Chicago, Illinois, April 18-21, 1994, pp. 348-357.

[Conte92] Thomas A. Conte, "Tradeoffs in Processor/Memory Interfaces for Superscalar Processors, Proceedings of the 25th Annual International Symposium on Microarchitecture, Portland, Or 1992.

[Cvet94] Zarka Cvetanovic and Dileep Bhandarkar, "Characterization of Alpha AXP Performance Using TP and SPEC Workloads, The 21st Annual International Symposium on Computer Architecture, April 18-21, 1994, pp. 60-70.

[Fark94] Keith I. Farkas and Norman P. Jouppi, "Complexity/Performance Tradeoffs with Non-Blocking Loads", ISCA-21, Chicago, Illinois, April 18-21, 1994, pp. 211-222.

[Farr94] Mathew Farrens, Gary Tyson, and Andrew R. Pleszkun, "A Study of Single-Chip Processor/Cache Organizations for Large Numbers of Transistors", ISCA-21, Chicago, Illinois, April 18-21, 1994, pp. 338-347.

[Gee93] Jeffrey D. Gee, Mark D. Hill, Dionisios N. Pnevmatikatos, and Alan Jay Smith, "Cache Performance of the SPEC92 Benchmark Suite", IEEE Micro, August 1993, pp. 17-27.

[Gray93] Jim Gray, Ed., "The Benchmark Handbook for Database and Transaction

Processing Systems”, Morgan Kaufmann Publishers, 1993.

[Gwen94] Linley Gwennap, “MIPS R10000 Uses Decoupled Architecture”, Microprocessor Report, Volume 8, Number 14, October 24, 1994, pp. 18-22.

[Henn90] John L. Hennessy and David A. Patterson, “Computer Architecture a Quantitative Approach”, Morgan Kaufmann Publishers, Inc, 1990.

[John91] Mike Johnson, "Superscalar Microprocessor Design", Prentice-Hall Inc, 1991.

[Joup90] Norman P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prfetch buffers”, Proc. 17th Annual Int’l Symposium on Computer Architecture (Cat. No. 90CH2887-8), IEEE Computer Society Press, Los Alamitos, CA, Seattle, May 28-31, 1990, pp. 364-373.

[Joup93] Norman P. Jouppi, "Cache Write Policies and Performance", ISCA-20, San Diego, California, May 16-19, 1993.

[Krof81] David Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization", ISCA-8, 1993 pp. 81-87.

[Kusk94] Jeff Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John L. Hennessy, “The Stanford FLASH multiprocessor”, Proceedings of the 21st International Symposium on Computer Architecture, pp. 302-313, April 1994.

[Mayn94] Ann Marie Grizzaffi Maynard, Colette M. Donnelly, and Bret R. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads", ASPLOS-VI, San Jose, CA, October 4-7, 1994.

[McLe93] Edward McLellan, "The Alpha AXP Architecture and 21064 Processor", IEEE Micro, June 1993, pp. 36-47.

[Rose95] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel, and Anoop Gupta, "The Impact of Architectural Trends on Operating System Performance", To Appear in The 15th ACM Symposium on Operating Systems Principles, Copper Mountain Resort, Colorado, Dec. 3-6, 1995.

[Rose95b] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta, “Complete Computer System Simulation: The SimOS Approach”, IEEE Parallel and Distrubuted Technology, Volume 3, Number 4, Fall 1995.

[MIPS94] MIPS Technologies, Incorporated, "R10000 Microprocessor Product Overview", MIPS Open RISC Technology, MIPS Technologies, Incorporated, October 1994.

[NEC94] NEC Corporation, "16M bit Synchronous DRAM, preliminary data sheet", NEC Corporation, March 1994.

[Oluk92] Kunle Olukotun, Trevor Mudge, and Richard Brown, "Performance Optimization of Pipelined Primary Caches", ISCA-19, Gold Coast, Australia, May 19-21, 1992, pp. 181-190.

[Przy88] Przybylski, S., M. Horowitz, and J. Hennessy, “Performance Tradeoffs in Cache Design”, Proceedings of the 15th Annual International Symposium on Computer Architecture, June 1988, pp.290-298.

[Rau93] B. Ramakrishna Rau and Joseph A. Fisher, "Instruction-Level Parallel Processing: History, Overview, and Perspective", Journal of Supercomputing, 7, 1993, pp. 9-50.

[Sohi91] Gurindar S. Sohi and Manoj Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors", ASPLOS-IV, Santa Clara, CA, April 8-11, 1991.

[SPEC95] SPEC, “SPEC Benchmark Specifications - 101.tomcatv”, SPEC95 benchmarks release, 1995.

[Toma67] Tomasulo, R. M., “An Efficient Algorithm for Exploiting Multiple Arithmetic Units.”, IBM Journal of Research and Development, Vol. 11 (January 1967), pp. 25-33.

[Uht86] Uht, A. K., “An Efficient Hardware Algorithm to Extract Concurrency from General Purpose Code”, Proceedings of the Nineteenth Annual Hawaii International Conference on System Sciences, 1986, pp. 41-50.

[Upto94] Michael Upton, Thomas Huff, Trevor Mudge, and Richard Brown, "Resource Allocation in a High Clock Rate Microprocessor", ASPLOS-VI, San Jose, CA, October 4-7, 1994, pp. 98-109.

[Wall93] David W. Wall, "Limits of Instruction-Level Parallelism", WRL Research Report 93/6, Western Research Laboratory, 250 University Ave., Palo Alto, CA, 94301, November 1993.

[Wilt94] Steven J. E. Wilton and Norman P. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches", WRL Research Report 93/5, Western Research Laboratory, 250 University Ave., Palo Alto, CA, 94301.

[Witc96] Emmett Witchel and Mendel Rosenblum, “Embra: Fast and Flexible Machine Simulation”, To appear in the Proceedings of ACM SIGMETRICS '96: Conference on Measurement and Modeling of Computer Systems, Philadelphia, PA, 1996.