The 18th Annual International Symposium on Computer Architecture. May, 1991.

Implementing a Cache for a High-Performance GaAs Microprocessor *

O. A. Olukotun T. N. Mudge R. B. Brown

Dept. Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, Mi 48109-2122

Abstract

In the near future, microprocessor systems with very high clock rates will use multichip module (MCM) packaging technology to reduce chip-crossing delays. In this paper we present the results of a study for the design of a 250 MHz Gallium Arsenide (GaAs) microprocessor that employs MCM technology to improve performance. The design study for the resulting two-level split cache starts with a baseline cache architecture and then examines the following aspects: 1) primary cache size and degree of associativity; 2) primary data-cache write policy; 3) secondary cache size and organization; 4) primary cache fetch size; 5) concurrency between instruction and data accesses. A trace-driven simulator is used to analyze each design's performance. The results show that memory access time and page-size constraints effectively limit the size of the primary data and instruction caches to 4KW (16KB). For such cache sizes, a write-through policy is better than a write-back policy. Three cache mechanisms that contribute to improved performance are introduced. The first is a variant of the write-through policy called *write-only*. This write policy provides most of the performance benefits of *sub-block placement* without extra valid bits. The second, is the use of a split secondary cache. Finally, the third mechanism allows loads to pass stores without associative matching.

Keywords—two-level caches, high performance processors, gallium arsenide, multichip modules, trace-driven cache simulation.

1 Introduction

The integration level of gallium arsenide (GaAs) technology is now high enough to support the fabrication of a RISC CPU and floating-point accelerator (FPA) on a single processor chip. However, it is not yet practical in GaAs to integrate primary cache with the CPU and FPA. The architectural design of the cache system for such a processor must, therefore, consider how chip-crossing delays between the processor and cache will impact system performance. Multichip modules (MCMs) have been developed in recent years to reduce the penalty associated with chip crossings in high-performance systems. In this packaging technology, bare integrated circuits are mounted in close proximity on a substrate having high-density interconnect. In systems using MCM packaging, partitioning must address not only which functions go on each chip, but also, which chips go on the MCM. The guiding principle is to place components on the MCM which, through low-latency communication with the CPU, will produce the greatest increase in system performance. The effects of different partitioning schemes can only be determined through simulation or experimentation.

Our purpose is to design a cache architecture for a 250 MHz microprocessor that is optimized for performance given specific technology constraints. The constraints are the area of the MCM, the speed, physical size and organizations of the primary and secondary cache SRAMs and the main memory latency time and transfer rate. Within these constraints we have evaluated the design of a number of two-level cache architectures. This design process is subdivided into five phases: primary cache size and associativity, primary data-cache write policy, secondary cache size and organization, primary cache fetch size and techniques that increase memory system concurrency. The design decisions for the first phase rely on the physical properties of both the MCM and GaAs technology. The successive designs rely on the decisions made in previous phases. The architectural performance data presented in this paper are the result of detailed trace-driven simulations of real benchmarks.

The following section presents a base architecture that serves as the starting point for our design study, and gives a brief overview of technological issues which bear on the cache architecture. Section 3 discusses the

*This work was supported by the Defense Advanced Research Projects Agency under DARPA/ARO Contract No. DAAL03-90-C-0028.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1991 ACM 0-89791-394-9/91/0005/0138 $1.50

simulation methodology used to evaluate alternative designs. Analysis of the base architecture with these simulation tools is reported in Section 4. In Section 5 we show that for the technology we are using 4 KW (16 KB) represents the best size for the primary instruction and data caches. Our study of primary-cache write policy shows a trade-off between write policy and secondary cache access time. A write-through policy has higher performance when the access time is less than 8 CPU cycles while write-back policy performs better for access times greater than 8 CPU cycles. We present a variant of a write-through policy called *write-only* which performs almost as well as subblock placement without the use of extra valid bits. Section 6 shows that splitting a large direct-mapped secondary cache, produces performance improvements and implementation benefits. Section 7 shows that the performance improvement from increasing the amount of concurrency in the memory system is small. Concluding remarks are made in Section 8.

2 Base Architecture

The base architecture is diagrammed in Fig. 1 and shows a single-chip CPU that integrates a pipelined instruction set processor and FPA based on the MIPS architecture [Kan87]. The figure also shows a two-level cache architecture comprised of a high-speed, low-latency primary (L1) cache and a much larger, high bandwidth secondary (L2) cache. The L1 cache, which is split into I-cache (instructions) and D-cache (data), and the L2 tags are to be implemented with 1K × 32-bit SRAMs that have a 3 nanosecond access time. Finally, the figure shows a memory management unit chip (MMU) and a write buffer (WB) chip. These, together with the CPU, are being designed in Vitesse's HGaAs III process.

The CPU has a critical path that limits the cycle time to just under 4 nanoseconds. However, the memory access paths to the L1 caches have the potential to increase this cycle time because they make two chip crossings. If the components are mounted on a conventional PCB, simple circuit models are sufficient to show that the cycle time will be dominated by the time lost in signal propagation between the chips. The effects of chip crossings can be reduced if the components of the base architecture are mounted on an MCM. Several properties of MCMs make them superior to PCBs in high performance systems. First, physical distances can be reduced because MCMs allow the direct bonding of unpackaged dies. Second, MCM features are only a factor of 10 larger than on-chip features. Typical line widths are 10–20 microns with pitches of 30–40 microns. This compares very favorably to PCBs where feature sizes on the order of 1000 microns are typical. Third, MCMs also allow larger pinouts because components can be

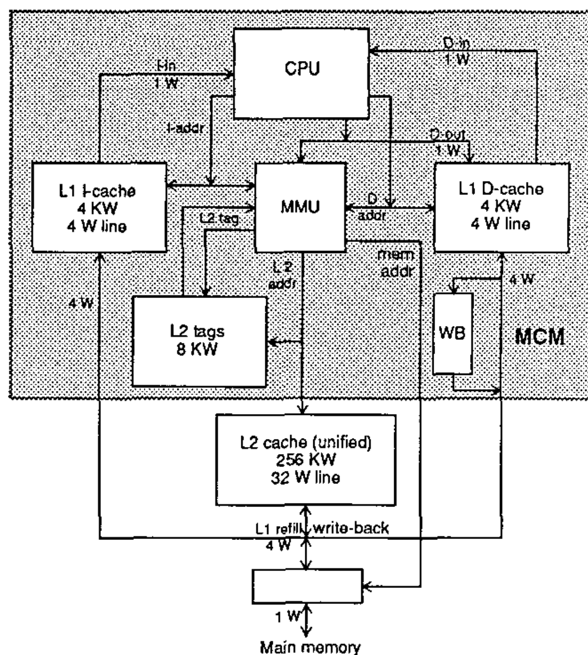

Figure 1: Baseline architecture.

mounted as flip-chip dies having interior pads. All three properties contribute to very high packaging densities that make low-latency, high-bandwidth inter-chip connections possible. In addition, superior electrical properties (low dielectric constant) and reduced packaging parasitics, coupled with short interconnect distances allow use of smaller, lower-power off-chip drivers.

Unfortunately, even when advanced packaging techniques are employed, inter-chip propagation delays are significant. Increasing primary cache size increases its area on the MCM and, consequently, inter-chip propagation delays. Furthermore, larger caches result in more loading for driver circuits. Both of these facts cause primary caches to have an access time that grows markedly with size. In our case, the inter-chip propagation delay and loading can contribute as much as 50% to the overall access time [Mud+91] and limit the cache size to a point where miss penalties can be significant. A second level of cache is employed to reduce this penalty [SL88, WBL89, Prz90].

As a starting point for this work, a base architecture is defined below:

- L1 I-cache (L1-I) is a direct-mapped 4KW cache with 4W lines and refill.
- L1 D-cache (L1-D) is a direct-mapped 4KW cache with 4W lines and refill.
- The L1 caches are write-back and require 1 cycle

to read, 2 cycles for a write hit and 1 cycle for a write miss.

- L2 is a unified write-back direct mapped 256KW cache with 32W lines and 32W refill.
- The refill path is 4W wide between L1 and L2 and is shared between I and D.
- The L1 miss penalty is 6 cycles (2 cycles communication delay and 4 cycles for the 4W transfer).
- The L2 cache is constructed from 8K $\times$ 8-bit BiCMOS SRAMs with an access time of 10 nanoseconds.
- The L2 miss penalties are 143 cycles for a clean miss and 237 for a dirty miss.
- A 4-deep, 4W wide write buffer connects the L1 D-cache with L2. The primary I- and D-cache both wait for this buffer to empty before processing a miss.
- The CPU, MMU, primary caches and secondary cache tags are mounted on the MCM.

A direct-mapped 4KW organization is chosen for the L1 cache sizes because the target machine's page size is 4KW and the operating system allows synonyms. This means that the untranslated address bits can index the cache without causing inconsistent synonyms. Hence, cache tag lookup does not require any bit translation and occurs in parallel with the virtual page number translation. This allows the smaller physical page number to be used as the tag instead of the larger virtual page number which must contain the process identifier (PID) (8 bits in our case).

The MMU contains the primary cache tags, the TLB logic, the write-buffer address queue, and the cache controllers for both levels of cache. It is possible to fit the tag memory for the 8KW primary cache entirely within the MMU if physical tags are used. Doing this makes tag checking and the control of primary cache misses simpler and faster. The tags take 40Kb of memory plus comparison logic. The TLB is organized as a 2-way set-associative 32 entry cache for instructions and a 2-way set-associative 64 entry cache for data. It requires 4Kb of memory. The write-buffer address queue is a 32-bit wide 4 entry shift register. A corresponding 128-bit wide 4 entry data queue is implemented in the WB chip. The MMU connects to the BiCMOS L2 SRAM through ECL compatible I/0 pads.

The 4W line size is a limitation imposed by the connector bandwidth of the MCM. The L1 caches are write-back. Write hits take 2 cycles to allow for tag checking before the write is committed. Following the lead of earlier work on two-level caches, we start with a unified L2 cache [SL88, WBL89, Prz90, TDF90]. The miss penalties for L2 are those of the R6020 system bus chip [Tho90] of the ECL-based MIPS RC6230 which will be used for prototyping. Waiting for the write buffer empty before fetching the data for a primary cache miss is necessary to keep the secondary cache consistent.

3 Simulator and Benchmarks

Computer performance is proportional to the average number of cycles required to execute an instruction. Commonly referred to as the CPI (cycles per instruction), it is expressed as,

$$\mathrm{CPI} = 1 + \frac{\texttt{CPU_stall_cycles} + \texttt{memory_stall_cycles}}{\texttt{instruction_count}}$$

where `CPU_stall_cycles` are due to multi-cycle instructions such as loads, branches, and floating-point operations, and `memory_stall_cycles` are the number of extra cycles spent fetching instructions and loading and storing data.

A trace-driven simulator has been developed to accurately count the number of cycles spent in the memory system. It is based on the MIPS suite of program performance analysis tools, `pixie` and `pixstats` [MIP88b]. The simulator is capable of modeling a wide variety of a two-level cache memory configurations. The effects of a multiprogramming environment are also modeled.

The first step in the simulation process is to use `pixie` to instrument the benchmarks that will provide the address traces. `pixie` takes an object file and augments it with instructions at basic block entry points and data reference instructions so that when the augmented object file is executed, it produces a trace of instruction and data reference addresses. To enable the simulator to switch among processes when voluntary system call instructions are executed, a system call file that contains the address of all system call instructions is generated for each benchmark. We pessimistically assume that all voluntary system calls cause context switches to occur. A separate cache simulator program is generated for each memory configuration. The resulting cache simulator runs efficiently because it only contains necessary code and all memory system parameters are constants. A full multiprogram cache simulation executes at the rate of 240,000 references per second on a MIPS RC3240 (a 15–20 MIPS system).

There are three parts to a cache simulation run: the file descriptor multiplexor, the benchmarks and the cache simulator. The file descriptor multiplexor uses UNIX pipes to map a trace output file descriptor of a benchmark program to an input file descriptor of the cache simulator. Each benchmark is mapped to a unique input file descriptor. Context switching between benchmarks is achieved by switching among the

input file descriptors from which traces are read. During initialization, the cache simulator reads a process configuration file and the system call files. The process configuration file specifies the number of processes that run concurrently (the multiprogramming level), and the order in which the processes are run. Each benchmark is counted as a single process. A hash table that contains a list of system call basic block entry points is created for each benchmark from its system call file. During simulation, a context switch is scheduled whenever the program counter matches an entry in a benchmark's system call hash table, or after the process' time slice has elapsed. The next process that runs is selected using a round-robin schedule. When a benchmark terminates the next benchmark in order is started. This continues until all the benchmarks have terminated.

The architecture supports process identifiers (PIDs) that are included as prefixes to virtual addresses so that each process has a distinct address space. This improves performance by eliminating the need to flush the caches and the translation-lookaside buffer (TLB) after every context switch [Aga88]. The simulator also models this feature. The virtual to physical mapping of addresses is performed using page coloring [TDF90].

The importance of realistic workloads for obtaining meaningful predictions of cache performance has been recognized for some time [Smi85]. More recently, the importance of long traces for obtaining accurate performance figures of large secondary caches has also been demonstrated [BKW90]. The simulation methods that were described above are as realistic as possible without including operating system kernel references at the context switch points and rely on the use of benchmarks that approximate a real workload (Table 1). Described in [MIP88a] they consist of a variety of C and FORTRAN programs. Executing the benchmark suite generates a total of about 2.5 billion memory references.

To include the effects of multiprogramming on various cache configurations, it is important to choose the values of multiprogramming level and process switch interval that are appropriate for a system as fast ours. To make these choices we used the base architecture and the cache simulator described above. The results are shown in Figs. 2 and 3. Experiments conducted as other cache architectures were explored showed these results to be fairly typical for the full range of architectures considered in this work. Fig. 2 shows that for a time slice of 500,000 cycles, performance degrades only slightly as the level of multiprogramming increases. The underlying mechanism for this effect appears to be the following: some of the lines brought into the cache by a process that is subsequently switched out will be evicted before the process is restarted. The number of lines that are evicted before a process restarts increases with the number of processes between restarts, i.e., as the level

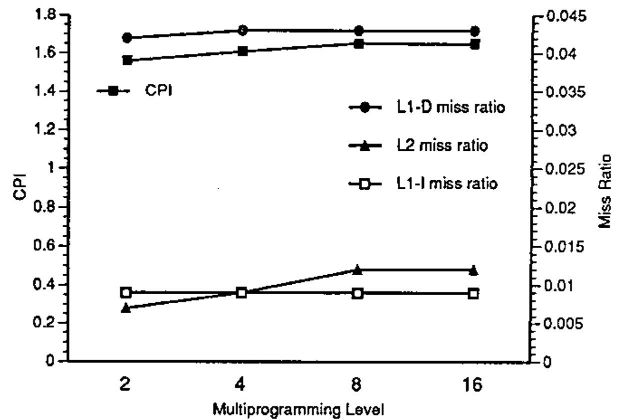

Figure 2: The effect of multiprogramming level on cache performance.

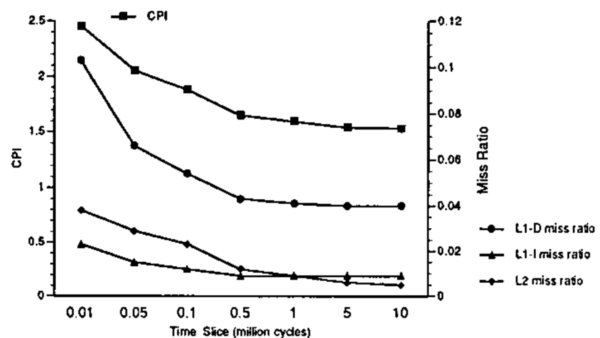

Figure 3: The effect of context switch interval level on cache performance.

of multiprogramming increases. The L1 caches are too small to show this effect in a pronounced way. For instance, the L1-I miss rate does not change and the L1-D miss rate changes by only 2%. However, the L2 cache is big enough to simultaneously contain lines from several processes and so its miss rate changes by 70%. Fortunately for overall performance, this is 70% of a very small number ($\approx 0.5\%$, see Fig. 2). In summary, for systems in the general range of those under investigation, cache performance is unaffected by multiprogramming levels of more than eight. Therefore we use a multiprogramming level of eight for our remaining studies.

Performance is improved significantly by increasing the time slice (see Fig. 3). This is due to the greater opportunity of reusing lines that are brought into the cache. It is clear that picking a time slice that is too short will result in poor cache performance [BKW90]. To guide the choice of a realistic time slice we examined the literature. Clark et al. have investigated the frequency with which context switches and interrupts occur on a VAX 8800 using a hardware monitor [CBK88]. They report an average of 7.7 milliseconds between con-

Benchmark	Description	Instructions (million)	Loads (% of inst.)	Stores (% of inst.)	# System calls
Matrix500	Operations on 500 × 500 matrices (S)	202.2	24.3	3.5	10
5diff	File comparison (I)	218.3	15.3	3.4	305
awk	Text string processing (I)	34.9	19.0	12.6	16
doduc	Monte Carlo simulation (D)	48.1	31.0	10.0	213
dhrystone 2.0	Integer benchmark (I)	53.6	25.7	10.8	2
espresso	Logic minimization (I)	119.0	19.9	5.6	8
gnuchess	Chess program (I)	488.2	19.6	7.2	148
grep	Pattern matching (I)	49.4	15.7	4.8	564
integral	Numerical Integration (D)	110.5	37.0	10.4	12
linpack	Linear equation solver (D)	4.0	37.4	19.7	10
LFK12	First 12 Livermore kernels (D)	275.5	29.3	10.9	3
nroff	Text formatting (I)	15.7	22.5	10.8	1701
small	Stanford small benchmarks (I/S)	16.7	19.9	8.8	0
spice2g6	Circuit simulator (S)	297.3	29.8	8.6	395
wolf33	Simulated annealing placement (I)	115.2	30.011	7.450	407
yacc	Parser generator (I)	96.9	19.647	2.413	24
Total		2145.9	24.1	7.3	3818

Table 1: List of benchmarks that were used to create a multiprogramming workload. Integer benchmarks are denoted by (I), single precision floating point benchmarks by (S), and double precision floating point by (D).

text switches. This time would translate into 1.9 million cycles for a computer with a 4 nanosecond cycle time. However, this represents a context switch interval that is too long; it ignores I/0 and timer interrupts which cause operating system kernel code to be executed, thus having a negative effect on cache performance similar to context switching. If we assume that I/0 devices and timer interrupts are unaffected by a shorter CPU cycle time, we could use Clark's figure of 0.9 milliseconds between any interrupt to take these interrupts into account. This time represents 225,000 4 nanosecond cycles. However, this context switch interval is too pessimistic because most context switches to the operating system kernel that are caused by interrupts switch back to the process that was interrupted. Such a process is likely to find a significant portion of its working set is still retained in a large secondary cache. This is illustrated by the lower L2 miss rate for a multiprogramming level of two in Fig. 2. As a compromise we chose a time slice of 500,000 cycles for our experiments. It results in an average of 310,000 cycles between context switches when all system call context switches are also included. It is interesting to note that faster machines may achieve lower cache miss rates because they execute more cycles between context switches.

4 Base Architecture Performance

The performance of the base architecture is shown in Fig. 4. Also shown is the performance loss breakdown from each of the components of the system as different gray levels. The horizontal axis at 1.238 CPI represents

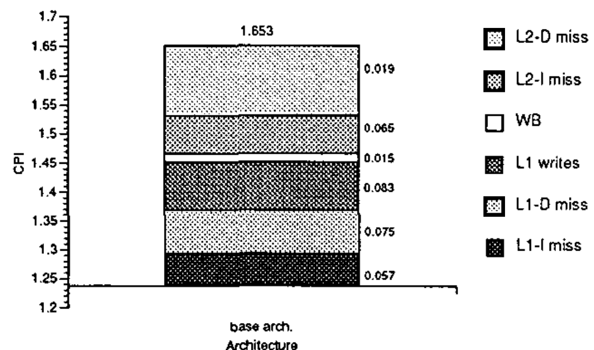

Figure 4: Performance losses of the base architecture.

the contribution of single cycle instruction execution, processor stalls resulting from load delays, branch delays and multicycle operations. The histogram above this axis is the contribution to CPI from the cache system and the focus of our discussion.

In the remainder of this paper architectural techniques that improve the performance of the base architecture by decreasing the value of CPI without increasing cycle time will be investigated. Throughout the exploration of alternatives we err on the side of architectures with simple implementations to avoid unexpected performance traps.

5 Primary Cache Size

We begin our investigation with the primary cache (L1) because it can determine the system cycle time. Here, the goal is to find a balance of size and orga-

nization tradeoffs that yields a maximum performance design.

The L1-I cache size is not limited by the page size constraint because it is a read only cache and so inconsistent synonyms cannot occur. Therefore, it is feasible to increase the L1-I cache size and thereby decrease the CPI. However, the additional SRAMs required to implement an 8KW cache (4 more for memory and 2 more for virtual tags) coupled with the additional time required to translate the virtual address will increase the access time enough to nullify the positive effects of a lower miss ratio [OBL+91] and produce an overall performance loss.

For a direct mapped L1-D cache, it is impossible to increase cache size beyond the page size without operating system support. One might consider a set-associative L1-D cache, but that would force the L1-D tags off the MMU chip. This additional tag access time added to the comparison time almost doubles system cycle time. Again, an increased cache size does not increase overall performance.

6 Write Policy

By adding together the contributions of 2 cycle writes (*L1 writes*) and write buffer cycles (*WB*) to the CPI in Fig. 4 it is determined that *writes* account for 24% of the performance loss in the memory system. The effect of write policy on system performance is studied in an attempt to reduce this loss. Four write-policies are considered, they are: the *write-back* policy used in the base-architecture, a *write-miss-invalidate* policy, a new policy called *write-only* and *subblock placement* [HP90]. The write-back policy uses the four entry deep write-buffer defined in the base architecture. The other three write-through policies use an one word wide, eight entry deep write buffer.

In the write-back policy used in this study, writes that hit in the cache take two cycles and those that miss take one cycle. *Write-allocate* is used in the event of a write miss. A modified line that is replaced is sent to a four-word deep write-buffer. In the write-miss-invalidate policy write hits take one cycle and write misses take two cycles. It is possible to complete write hits in a single cycle because the tag is checked while the data is written to the cache. If a miss is detected, a second cycle is used to invalidate the corrupted line. No previously written information is lost by doing this because all writes are sent to the write buffer. Our new write-only policy modifies write-miss-invalidate by updating the tag on a write-miss and marking the line as write-only. This allows one cycle completion of subsequent writes to the line. All reads that map to a write-only line miss and cause the line to be reallocated. In subblock placement each tag has four extra valid bits; one for each of the four words in the line. A write-miss

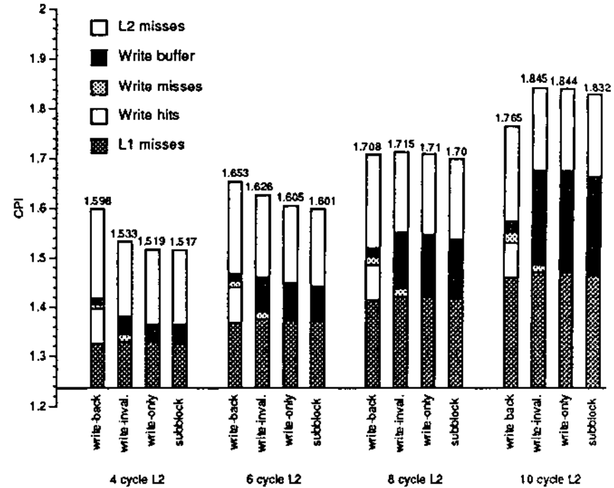

Figure 5: Write policy-L2 access time tradeoff for the base architecture.

causes the address portion of the tag to be updated in the next cycle. If the write was a word-write, the corresponding valid bit is turned on and all other bits are turned off. Subsequent word writes to the line update the valid bits in one cycle. However, partial word writes to the line do not update the valid bits.

The performance of the base architecture using these four write-policies is shown in Fig. 5 for L2 access times of two to ten CPU cycles. These access times assume a two cycle latency to account for L2-tag checking and communication delay between the L1 and L2 caches. Single reads and writes of L2 take the full access time. However, a stream of writes may overlap one or both cycles of latency. Fig. 5 shows that for L2 access times of less than 8 cycles write-through policies achieve higher performance, while for L2 access times greater than 8 cycles the write-back policy achieves higher performance. The reason is that for a 4KW, cache writes hit in the cache most of the time. Since writes make up a 0.0725 fraction of instructions, the constant 0.071 loss in CPI from 2-cycle-write-hits (*Write hits* in Fig. 5) shown by the write-back policy indicates a hit rate of 98%. In contrast, the write-through policies lose significantly less performance due to two-cycle-write-misses because the write-miss rate is only 2%. However, write-through polices waste many more cycles on average waiting for the write buffer to empty before fetching the data for a L1 read miss than a write-back policy. The number of cycles it takes to empty the write-buffer is determined by the effective access time of the secondary cache. This assumes that changes in L2 cache size can be related to changes in effective L2 cache access time [Prz90]. At some value of effective L2 access time the extra time spent waiting for the write-buffer in the write-through

policies will be greater than the cost of two-cycle-write-hits in the write-back policy. This results in a trade-off between write policy and effective L2 cache access time. We note that the exact nature of this trade-off depends on the L1 read miss ratio and therefore on L1 size. However, the L2 access time at which a write-back policy becomes the better choice grows with L1 cache size because larger L1 caches have fewer read and write misses.

Fig. 5 shows that for the region of L2 access time in which a write-through policy is most effective (4 and 6 cycles) write-only performs almost as well as sub-block placement. The reason for this is that most of the performance gain (over 80%) from subblock placement over write-miss-invalidate comes from writes misses that cause subsequent writes to hit. The extra performance gain from read hits that would otherwise have missed is less than 20%. Write-only requires 3Kb less tag memory than subblock placement for the base architecture. Furthermore, write-only does not need the ability to read and write the tag RAM in the same cycle. For this reason we will use the write-only write-through policy in the remainder of this paper.

Using a write-through cache increases the number of slots in the write-buffer by a factor of two (8 deep instead of 4 deep), but decreases its width from four words to one word. The accompanying factor of four reduction in I/O requirements, from 256 pins to 64 pins, enables us to put the write-buffer inside the MMU chip. Thus, a write-only policy provides higher performance and a cheaper and simpler implementation than a write-back policy for the L2 access times of the base architecture.

7 Secondary Cache

The effects of secondary cache size and organization on performance are investigated by looking at four candidate cache organizations: unified direct-mapped, unified two-way associative, split direct-mapped and split two-way associative. A split cache is logically partitioned into instructions and data. It is assumed that making a cache associative adds one CPU cycle increasing the access time of the cache from 6 to 7 cycles or 16.7%. Fig. 6 shows the performance of these four cache organizations. This figure shows that splitting improves the performance of direct-mapped caches of 64KW or more. The performance benefit of splitting is not evident for two-way associative caches until the cache size reaches 512KW. The benefits of splitting large caches come without an increase in cache access time or cache access bandwidth because splitting can be implemented by using the high-order bit of the cache index to interleave between the instruction and data halves of the cache.

Two processes access the secondary cache: instruction fetching and data accessing. These two processes never share address space, but in a direct-mapped cache, they can interfere with one another because of mapping conflicts (two memory locations mapped to the same line). This is a feature of direct mapped caches that degrades their performance [Hil87]. We have shown that these conflicts can be reduced without affecting access time by dividing the cache into separate instruction and data portions. Others have shown that the miss ratios for split direct-mapped primary caches are no worse than those of unified caches [WBL89]. Table 2 shows that for large secondary caches, splitting actually improves the miss ratio. This improvement is expected to increase for even larger cache sizes because a greater fraction of misses are due to conflicts.

To further investigate the effect that splitting the L2 cache has on performance the speed-size tradeoff curves for the L2 instruction cache (L2-I) and L2 data cache (L2-D) are shown in Figs. 7 and 8. The effect of writes on L2-D is ignored in order to simplify the comparison between L2-I and L2-D. The data in the figures are the result of varying the speed and size of the L2-I and L2-D caches from the base architecture. Both sets of curves show the same trend; the marginal perfor-

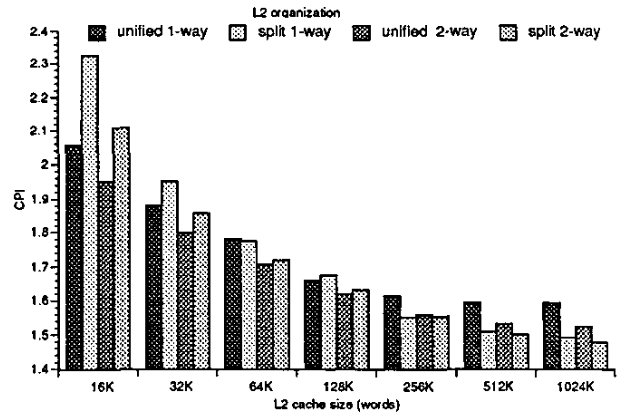

Figure 6: Performance of various L2 sizes and organizations. 1-way associative caches have a 6 cycle access time; 2-way set-associative caches have a 7 cycle access time.

size (words)	unified 1-way	unified 2-way	split 1-way	split 2-way
16K	0.0335	0.0269	0.0489	0.0364
32K	0.0240	0.0181	0.0273	0.0221
64K	0.0186	0.0129	0.0177	0.0143
128K	0.0133	0.0078	0.0121	0.0093
256K	0.0112	0.0048	0.0072	0.0051
512K	0.0102	0.0034	0.0051	0.0025
1024K	0.0102	0.0031	0.0042	0.0013

Table 2: L2 miss ratios for the sizes and organizations of Fig. 6.

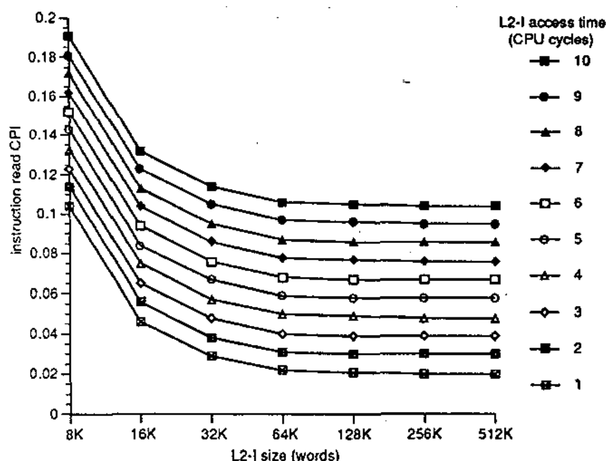

Figure 7: The L2-I speed-size tradeoff with a 4KW L1-I.

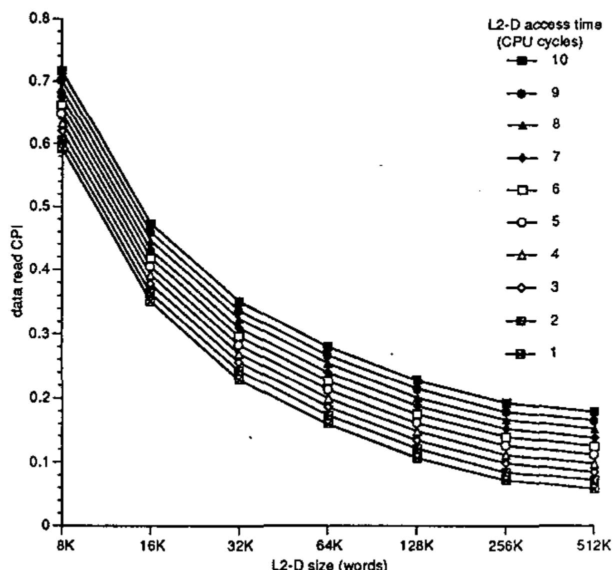

Figure 8: The L2-D speed-size tradeoff with a 4KW L1-D.

mance increase due to increasing cache size is smaller for larger cache sizes. However, the exact shape and level of performance of the L2-I and L2-D caches is quite different. The L2-I cache performance varies from 0.19 CPI to 0.02 CPI and the curves are fairly flat for cache sizes larger than 64KW. Whereas the L2-D cache performance varies from 0.72 CPI to 0.06 CPI and is still decreasing at 512KW. From this data we conclude that the optimum data cache size is roughly 8 times as large as the optimum instruction cache. Two-way associative split L2 caches show the same trends, although the curves are shifted downward due to the lower miss rates and the L2-D cache performance improves more than the L2-I cache.

The speed-size tradeoffs for secondary instruction and data parts of a split secondary cache are radically different. To take advantage of this fact we should implement the L2-I cache in a faster, less dense technology than the L2-D cache. To do this, a 32KW L2-I cache is implemented from the same 1K × 32-bit chips as the L1 caches. The access time of this cache is two CPU cycles. The 256KW secondary cache from the base architecture is made into a 256KW L2-D cache. However, the L2-D cache is not made 2-way set-associative because its 0.012 decrease in CPI is too small a performance benefit for the extra multiplexing, interconnect, and MMU pins necessary to implement it.

The performance gain describe above is shown by the difference between the first and second columns of Fig. 9. A substantial 34% increase in performance is achieved. The contribution of the memory system to CPI is now 0.242. If we exchange the sizes and access times of L2-I and L2-D, the CPI increases by 21% to 0.293. It is clear that L2-I should be placed on the MCM while L2-D should be placed off of the MCM.

8 Primary Cache Fetch Size

With the latency and transfer rates between L2 and L1 fixed for both instructions and data we can investigate the effect that fetch size has on performance [Smi87, Prz90]. The L1-I latency is two cycles and the transfer rate is four words every cycle. For L1-D the latency is six cycles and the transfer rate is four words every cycle.

Performance is maximized when the fetch size and line size of L1-I are both eight words long. The optimal L1-D fetch and line size is also eight words. Any further increase in the fetch sizes of either L1-I or L1-D will decrease performance. The total improvement in performance of lengthening the L1-I and L1-D fetch size to 8 words is a 0.026 decrease in CPI. This is shown by the difference between the second and third columns in Fig. 9. As a consequence of increasing the L1 line size by a factor of two the size of the L1 tags on the MMU chip has been reduced from 40Kb to 20Kb.

9 Memory System Concurrency

With the six optimizations we have implemented so far, memory performance has been improved by 48% and total performance by 12% as measured in CPI. In this section we will evaluate techniques that improve performance by allowing more concurrency between memory accesses.

With a split L2 cache instruction and data accesses are completely independent. Thus after an L1-I miss it is possible to refill the L1-I cache from the L2-I cache while the write-buffer continues to empty into the L2-D cache. This provides a decrease in CPI of 0.011 as

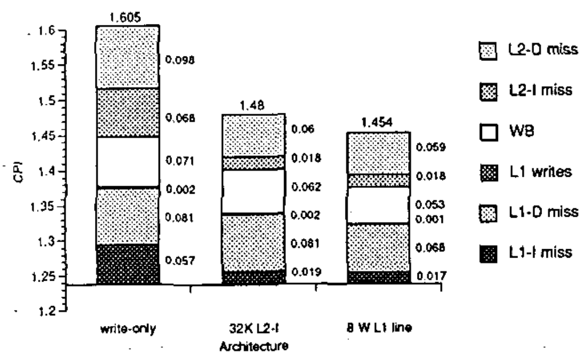

Figure 9: The performance improvement from adding a 32KW 2 cycle L2 on the MCM and from optimizing L1 fetch size.

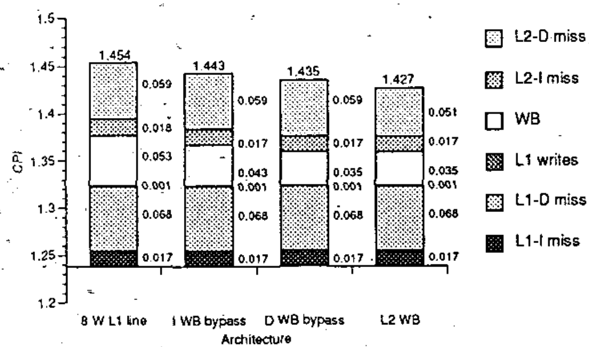

Figure 10: The performance improvement gained from adding more concurrency to the memory system.

shown in the second column of Fig. 10.

To allow data-reads to bypass data-writes in the write-buffer usually requires that all eight entries of the write-buffer associatively match the address of the missed line. If a match is made, then all entries ahead, including the matched entry, must be flushed to keep the state of the L2-D cache consistent. However, if an extra dirty bit is added to the L1-D tags. The cache need only be flushed when dirty lines are replaced in the cache. No associative matching is needed. This scheme works because the write-only policy ensures that all writes allocate a line in the L1-D cache. The write-buffer can only contain parts of dirty lines. Therefore, to keep the L2-D cache consistent it is only necessary to flush the write-buffer when these lines are replaced. Experiments show that this scheme achieves 95% of the performance increase of associative matching. However, this performance increase is very modest; only a 0.008 decrease in CPI. This performance is shown in Column *D WB bypass* in Fig. 10.

At this design point, the largest component of performance loss is that of L2-D dirty misses. Currently, when

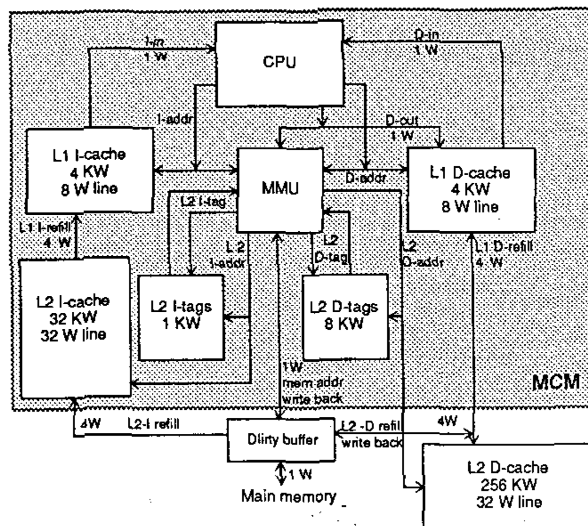

Figure 11: Optimized architecture.

a L2-D dirty miss occurs the dirty line is written to main memory and then the requested line is read from main memory. To improve this situation a single 32 word line write-buffer or dirty buffer can be added to L2-D cache. Now it is possible to read the requested line before the dirty line is written. However, the performance improvement from doing this is only a 0.008 decrease in CPI. It is shown in Column *L2 WB* of Fig. 10.

The total performance improvement from adding concurrency to the system is a 0.027 decrease in CPI. It is clear that increasing concurrency in the memory system in these limited ways adds much less to the performance of the memory system than the basic cache size, organization and speed optimizations discussed in the previous sections. In fact, it is questionable, whether the last two optimizations should be implemented at all given the increase in control logic and memory required.

The final three optimizations bring the total memory system performance improvement to 54.5% and total system performance improvement to 13.7%. The optimized architecture is diagrammed in Fig. 11

10 Conclusions

We have developed a cache simulator capable of modeling multiprogrammed workloads. The simulator was used to select appropriate values for multiprogramming level and time-slice, and to evaluate and refine the design of a two-level cache architecture our high-performance MCM-based microprocessor. The simulator provides detailed cycle counts that enabled us to evaluate a number of design alternatives based on total system performance. Our experiments show that a new write-only policy is better than a write-back policy and almost as good as subblock placement for secondary

cache access times of eight CPU cycles or less. Our experiments also show that there is a performance advantage to be gained from logically splitting the secondary cache into instruction and data parts. Performance is further enhanced if the cache is physically partitioned with a 32KW two cycle access time secondary instruction cache on the MCM and a 256KW six cycle access time secondary data cache off of the MCM. This partitioning is dictated by the difference between the speed-size trade-offs of secondary instruction and data caches and provides a significant increase in performance. Relatively less performance is gained by using extra hardware to add concurrency to memory accesses through the use of a dirty buffer, or through data-read conflict checking in the write buffer. Finally, this study has demonstrated that cache simulations must be tied to specific technological implementations in order to yield meaningful results.

Acknowledgments
We thank MIPS Computer Systems for supporting this project with technical assistance under a special licensing arrangement. We also gratefully acknowledge the assistance of Vitesse Semiconductor Corp., Seattle Silicon Corp., and Mentor Graphics Corp. Finally, we thank the anonymous reviewers for suggesting we look at subblock placement and David Nagle for his help in preparing the final manuscript.

References

[Aga88] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming.* Kluwer Academic Publishers, 1988.

[BKW90] A. Borg, R. E. Kessler, and D. W. Wall, "Generation and analysis of very long address traces," in *Proc. of the 17th Annual International Symposium on Computer Architecture*, pp. 270–279, June 1990.

[CBK88] D. W. Clark, P. J. Bannon, and J. B. Keller, "Measuring vax 8800 performance with a histogram hardware monitor," in *Proc. of the 15th Annual International Symposium on Computer Architecture*, pp. 176–185, June 1988.

[Hil87] M. D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, 1987.

[HP90] J. L. Hennessey and D. A. Patterson, *Computer Architecture A Quantative Approach.* San Mateo, California: Morgan Kaufman Publishers, Inc., 1990.

[Kan87] G. Kane, *MIPS R200 RISC Architecure.* Englewood Cliffs, New Jersey: Prentice Hall, 1987.

[Mud+91] T. N. Mudge et al., "The design of a microsupercomputer," *Computer*, vol. 24, Jan. 1991.

[MIP88a] MIPS Computer Systems, Inc, *Performance Brief CPU Benchmarks*, Oct. 1988.

[MIP88b] MIPS Computer Systems, Inc, *RISCompiler Languages Programmer's Guide*, Dec. 1988.

[OBL+91] O. A. Olukotun, R. B. Brown, R. J. Lomax, T. N. Mudge, and K. A. Sakallah, "Multilevel optimization in the design of a high-performance GaAs microcomputer," *IEEE J. Solid-State Circuits*, vol. 26, May 1991. (to appear).

[Prz90] S. A. Przbylski, *Cache and Memory Hierarchy Design.* San Mateo, California: Morgan Kaufman Publishers, Inc., 1990.

[SL88] R. T. Short and H. M. Levy, "A simulation study of two-level caches," in *Proc. of the 15th Annual International Symposium on Computer Architecture*, pp. 81–88, June 1988.

[Smi85] A. J. Smith, "Cache evaluation and the impact of workload choice," in *Proc. of the 12th Annual International Symposium on Computer Architecture*, pp. 64–73, June 1985.

[Smi87] A. J. Smith, "Line (block) size choice for CPU cache memories," *IEEE Trans. Computers*, vol. C-36, pp. 1063–1075, Sept. 1987.

[TDF90] G. Taylor, P. Davies, and M. Farmwald, "The TLB slice— a low-cost high-speed address translation mechanism," in *Proc. of the 17th Annual International Symposium on Computer Architecture*, pp. 355–363, June 1990.

[Tho90] M. Thorson, "ECL bus controller hits 266 Mbytes/s," *Microprocessor Report*, vol. 4, pp. 12–13, Jan. 1990.

[WBL89] W.-H. Wang, J.-L. Baer, and H. M. Levy, "Organization and performance of a two-level virtual-real cache hierarchy," in *Proc. of the 16th Annual International Symposium on Computer Architecture*, pp. 140–148, June 1989.