A Simulation Study of Two-Level Caches[1]

Robert T. Short
DECwest Engineering
14475 NE 24th Street
Bellevue, WA 98007

Henry M. Levy
Department of Computer Science
University of Washington
Seattle, WA 98195

Abstract

We report on a trace-driven simulation study to examine the effect of a two-level cache hierarchy in uniprocessors. A simulation model of a multiple-cycle-per-instruction processor was constructed to estimate the total cycles required to execute a synthetic benchmark. Results show that a second-level cache can be used to increase system performance when main memory access times are large relative to CPU cycle time. For example, the addition of a 4-cycle, 64K second-level cache following a 1-cycle, 8K first-level cache increases performance by 15 percent when used in a system with a 15-cycle primary memory. Second level caches are shown to be particularly effective when used behind small on-chip caches; adding an 8K second-level to a 1K first-level increases performance by 26 percent, assuming similar parameters. We also evaluate the performance impact of different write strategies and separate I and D caches.

1 Introduction

Since its 1968 introduction in the IBM 360/85 [17], the cache memory has become commonplace in computer architectures. The power of the cache derives from its high performance impact and low relative cost; the addition of a cache can easily double the performance of a system by greatly reducing effective memory access time. Cache behavior has been extensively studied through trace-driven simulation [23, 19, 13, 20], modelling [5, 25], and direct measurement [7, 9].

Caches provide an additional level in the system memory hierarchy. In this paper we examine the performance impact of adding a third level to this hierarchy through the use of a two-level cache. Two level caches have been described in the literature [22, 12, 4, 24] but have rarely been used. The one notable exception is the FACOM-380/382 [14] which has a fast 64K local cache and a second-level cache of 128K–512K bytes. Smith [19] initially discounted the feasibility of multilevel caches on the grounds that the ratio of memory to cache speed is too small for a third level; more recently, he suggested that multilevel caches would become more common [21].

Several trends in current technology have increased the viability of a two-level cache, for example:

1. Technology is yielding impressive reductions in processor speed, but memory speed has not kept pace. Therefore, the disparity between processor and memory speed is increasing.

2. The appetite for memory is rapidly increasing and primary memories of 100s to 1000s of megabytes will be common. Larger memories typically have larger access times because of packaging and physical constraints.

3. As on-chip densities increase, it becomes possible to include small on-chip instruction and/or data caches [15, 1, 6, 11, 3]. These caches will need to be backed up by larger second-level caches. As one example, the MicroVAX 3500 has a 1K on-chip cache and a 64K second-level cache.

4. Shared memory multiprocessors require local caches to reduce bus contention. A second-level global cache can help to further reduce access time on cache misses [10, 24].

For these reasons, we believe that two-level caches will become commonplace within the next several years.

This paper describes a simulation study of two-level caches in uniprocessors. While the goal of most cache simulations is to determine hit ratios, we have relied on a detailed CPU model to produce a more detailed performance analysis. That is, in addition to hit ratios, the simulation estimates program performance in CPU cycles based on our CPU model. The need for using such a model is dictated by the nature of a two-level cache: large changes in hit ratio at the second level (further from the CPU) may have only a small impact on program performance. Therefore, our simulation determines CPU cycles as well as hit ratios for the synthetic stream that it executes. The following section describes the models we examined and the methodology used for the simulation. We then describe the structure of the simulator and finally present results.

2 Cache Models and Methodology

For this study we chose to examine several system organizations, including (a) a baseline system with a single cache, (b) a system with a two-level cache, and (c) a two-level system with separate local instruction and data caches. Figure 1 shows the basic two-level cache organization with a single combined instruction and data cache, from which most of our results are reported. We refer to the processor-local cache as the L1 cache and the second-level cache as the L2 cache. At a high level, the components of this system operates in the following way:

- The instruction fetch unit contains an instruction register and a prefetch buffer into which the next instruction is read while the current instruction is executing. When the prefetch buffer is empty, the instruction fetch unit sends an address to the cache and waits for the instruction to return. This provides a simple model of instruction prefetch.

- The instruction execution unit fetches operands and stores results. The instruction execution unit waits until an instruction is ready in the instruction register, begins fetching operands from the cache, executes the instruction, and then stores results.

[1]This work was supported in part by the National Science Foundation under Grant No. CCR-8619663 and by DECwest Engineering.

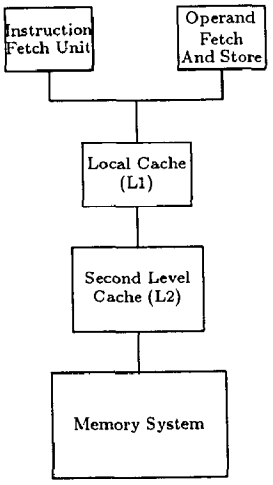


Figure 1: Basic Two-Level Simulation Model

- The L1 cache receives addresses from the prefetch and returns instructions either from the cache or from the next level of the memory hierarchy. The cache also receives addresses from the execution unit and reads or writes operands, again from the cache or from the next level of the hierarchy. The handling of writes varies with different write algorithms. If separate L1 instruction and data caches are present, they respond to the instruction fetch and instruction execution units, respectively.

- The L2 cache receives addresses from the L1 cache (or caches) and reads or writes operands from its storage or from the primary memory system. The handling of writes varies with different write algorithms.

- The bus is a half-duplex datapath connecting the caches to the memory system. Devices on the bus must arbitrate for bus ownership before commands or data can be sent.

- The primary memory consists of a number of interleaved memories. Simulation parameters include the interleaving factor, access time, and cycle time of main memory.

As previously stated, the goal was to estimate the total cycles required to execute a synthetic instruction stream. To do this, we defined the sequential operations required for the processing of each instruction: instruction fetch, operand fetch, instruction execution, and results storage. Some overlap occurs — for example, instruction prefetch is overlapped with current instruction execution — and this is handled by the model. Times were assigned to each of the possible operations modelled by the simulation, as defined in Table 1. Some times are fixed and some are parameters of the model. As shown in the table, the principal parameters we wished to vary were the L2 cache and main memory access times. The final operation, result write, is shown as overlapped because, although it uses cache bandwidth, it does not impede the progress of the following instruction until that instruction attempts an operand read.

These timings do not represent any specific computer and are meant to be realistic for the canonical models used. They more closely represent a CISC architecture than a RISC architecture because of the extra cycles required for execution and address generation. For example, Figure 2 shows the timing for some typical instruction executions through our model.

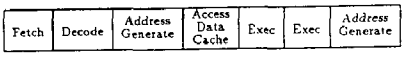
Operation	Cycles
instruction cache access	1
instruction decode	1
operand address generation	1
data cache access	1
address transfer on bus	1
L2 cache access	var
primary memory access	var
transfer memory to L2 cache	1
transfer L2 cache to L1 cache	1
instruction execution	2
generate result address	1
result write	overlapped

Table 1: Simulation Model Timing Parameters

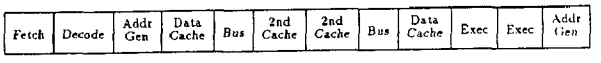
Figure 2a shows an instruction with a hit at the first level cache. In this case, an instruction takes 7 cycles through our model. For comparison, the average execution of an instruction takes about 10.6 cycles on a VAX-11/780 and 8.4 cycles on a VAX 8800 [8]. In Figures 2b and 2c we see the effect of L2 cache hits and misses, respectively.

Our simulation program is driven by a stream of instruction trace data obtained through the ATUM microcode trace facility on the VAX 8200 [2]. ATUM is capable of recording up to 400,000 consecutive references at a time. The advantage of microcode-generated traces over traces collected through software emulation is the inclusion of operating system as well as user instructions in the data. We used a selection of ATUM traces from scientific and system programs executed on the VAX/VMS operating system. These included simulations, circuit analysis programs, the Pascal compiler, and the VMS assembler.

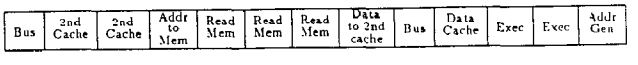
Instead of measuring the execution time of several individual benchmarks, we used the ATUM traces to construct a single composite "multiuser" benchmark. This gives us an estimate of the overall performance impact of the cache, including the effect of context switching. Our composite file consists of fifty thousand references from one source, followed by fifty thousand references from another source, and so on. The total composite trace consists of over 4.3 million instructions. These instructions generate 6.2 million read references (including 4.3 million for instruction reads) and 631,000 write references.



a. Instruction with data cache hit.



b. Instruction with second-level cache hit and second-level access time of two cycles.



c. Instruction with second-level cache miss and memory access time of three cycles, starting after data cache miss.

Figure 2: Example Instruction Timings

The addresses included by ATUM in the trace file are VAX virtual addresses. Because we are assuming a physically addressed cache, and it is possible that different process virtual addresses map to the same physical address, we required a scheme to map virtual addresses to physical addresses. This translation cannot be done exactly because ATUM does not record page table entries. However, because these programs were recorded under VAX/VMS and addressing under that system is deterministic, it is possible to differentiate in most cases addresses that are private from those that are shared by all processes. VAX virtual address space is divided into two parts: system space and process space [16]. All of system address space and some parts of the process-local address space are known to be shared.

To perform the virtual to physical address translation, each of the individual traces was treated as a separate independent process which may share some of its address space with other processes. Each process is assigned a unique number and this number is appended to the high end of its virtual addresses. Each process's memory is thus given a unique portion of the system-wide physical address space. Similarly, shared segments are given a unique portion of this address space and process virtual addresses to shared memory are mapped to the appropriate common regions.

3 Simulation Results

This section reports the results of executing our simulation model on the data stream described above. Our simulation model is fairly complex and permits many parameters to be varied. We have chosen to fix a number of the parameters and to concentrate primarily on those which we consider most relevant, namely the sizes of the L1 and L2 caches. In general, we have examined systems with primary memory access times that are large compared to cycle time. We have also made a number of simplifying assumptions. First, the line sizes of the L1 and L2 caches are identical (4 bytes). We examined variations in fill size (i.e., the number of cache lines read on each miss) but don't report here results for all combinations of sizes (more data is available in [18]). Second, we assume only direct mapped organization. Third, we assume no write allocate on write misses, and only limited write buffering.

Following subsections show simulation results given our model, parameters, and trace data. Obviously, changes in model (e.g., using a lockup-free cache), parameters (e.g., different line size or average cycles per instruction), or workload would cause a change in the results.

3.1 Hit Ratios

Figure 3 shows hit ratios for a single-level cache of various sizes. These hit ratios are consistent with those reported in a study of VAX cache behavior [7]. Of course, the hit ratio for a single-level cache will be the same as the hit ratio for the L1 cache in our two-level model. The figure also shows the effect of using 1-, 2-, and 4-line cache fills.

The total variation in cache hit ratio when increasing cache size from 8K bytes to 512K bytes is less than 10 percent. For a memory access time of 15 cycles, this difference in hit ratio accounts for almost a 50 percent increase in performance. The change in fill size can also be significant, particularly with small cache sizes; increasing fill size from 1 to 2 lines produces a 10 percent performance increase for an 8K byte cache.

Figure 4 shows the hit ratio of the L2 cache in our two-level cache system. As the figure shows, varying the size of the L2 cache has a significant effect on hit ratio, and our simulation shows L2 hit ratios ranging from 20 to 90 percent. Fill size can also affect the hit ratio of the L2 cache. For example, using an 8K L1 cache, increasing the L2 fill size from 2 to 4 lines increases its hit ratio by 7 to 20 percent, depending on its size.

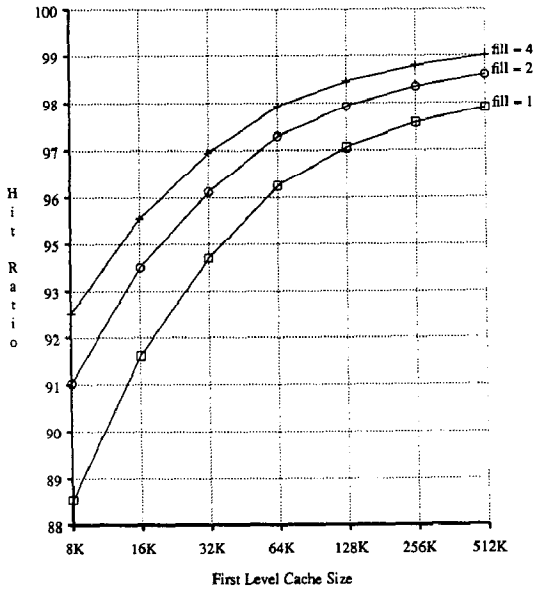


Figure 3: Hit Ratio of First Level Cache

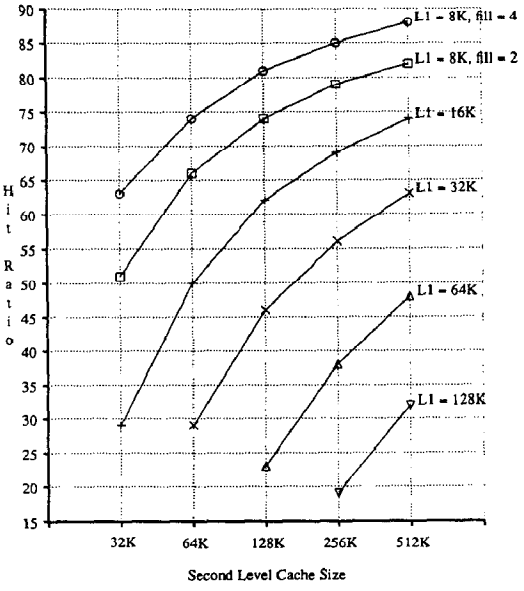


Figure 4: L2 Cache Hit Ratios

As would be expected, hit ratios of L2 caches are relatively low because the L2 cache sees a reference stream consisting of only misses to the L1 cache. This is true even for large L2 caches; with our trace data, an L2 cache of 512K bytes has a hit ratio of less than 75 percent when placed behind a 16K L1 cache, and less than 65 percent when placed behind a 32K L1 cache. As the L1 cache size increases, its hit ratio increases, and the hit ratio of the L2 cache decreases. This is shown more dramatically in Figure 5. Here we see that a 64K L2 cache suffers a one third decrease in hit ratio when its L1 cache is increased from 8K to 16K bytes. A 128K L2 caches suffers a 23 percent decrease for the same L1 size change.

3.2 Performance

Figure 6a shows the performance effect of increasing L2 cache size for various L1 cache sizes. It is interesting to note that while increasing the size of the L2 cache has a substantial impact on its hit ratio, the hit ratio change may have little effect on the real system-level performance. For example, consider the hit ratio and performance of an 8K L1 cache with 4-line fill and different L2 cache sizes. Increasing the size of the L2 cache from 32K to 512K increases its hit ratio from 63 to 88 percent — a 25 percent improvement. This improved hit ratio decreases the runtime from 30 million cycles to 28 million cycles, a performance improvement of only 7 percent. The reason for the disparity is straightforward; most memory requests are satisfied at the L1 cache, and only the small percentage that miss are sent to the second level. For example, if the L1 cache has a 90 percent hit ratio, only 10 percent of the requests are passed on to the L2 cache. Even if the L2 cache has a hit ratio as low as 50 percent, only 5 percent of memory requests will be passed to primary memory. In this case, improving the L2 hit ratio can at best improve on this 5 percent of requests going to primary memory. Of course, for primary memories with large access times, the improvement may be worthwhile.

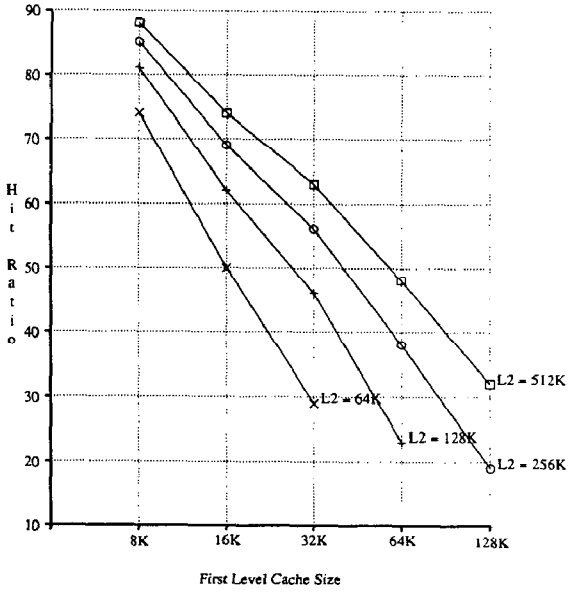
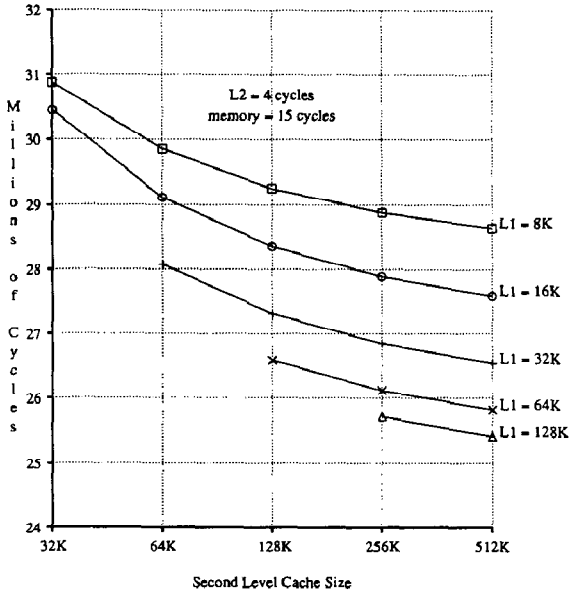
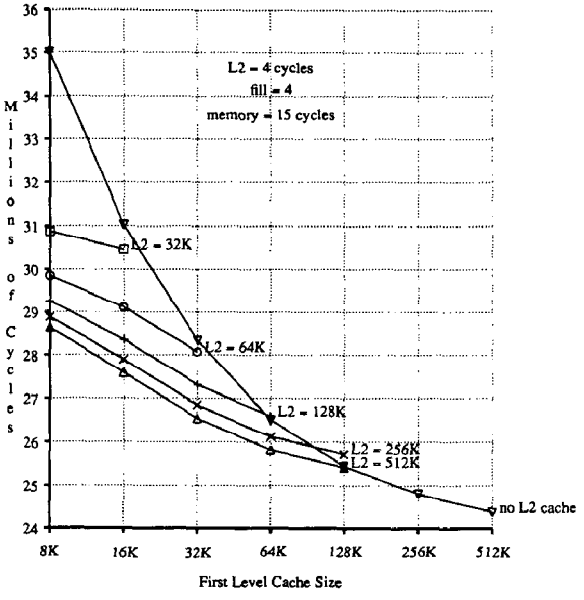


Figure 5: Effect of L1 size on L2 hit ratio.

It is clear from Figure 6a that a larger L2 cache will always outperform a smaller L2 cache. On the other hand, an L2 cache is *not* always beneficial. This can be seen in Figure 6b which shows the same two-level data plotted as a function of L1 cache size. Compared to a system with a one-level 8K cache (the top line in Figure 6b), adding a 32K L2 cache improves performance by almost 12 percent, while adding a 64K L2 cache shows a 15 percent improvement. However, adding a 128K L2 cache to a system with 64K L1 cache has no impact, and adding a 256K L2 cache to a system with 128K L1 actually degrades performance.



(a)



(b)

Figure 6: Performance of Two-Level Caches

The existence of an L2 cache can degrade overall system performance because it increases the total memory access time for accesses that miss. This becomes more of a problem as the L1 cache size increases and the L2 hit ratio decreases. If the L2 hit ratio is less than 50 percent, more than half of the accesses to the L2 cache will take an additional penalty in getting to primary memory. Starting the L2 lookup and the primary memory access in parallel will alleviate this problem but at the cost of wasted bus and memory bandwidth.

Figure 7 illustrates the effect of increasing the L2 cache access time for an L2 size of 128K. L1 sizes of 8K, 16K, and 32K are shown with various primary memory access times. An increase in L2 cache access time causes a linear increase in execution cycles with a greater slope for smaller L1 sizes. Obviously the smaller the L1 cache, the lower the L1 hit ratio, and the greater the effect of a slower L2 cache. Increasing the memory access time for a particular configuration results in a larger y intercept.

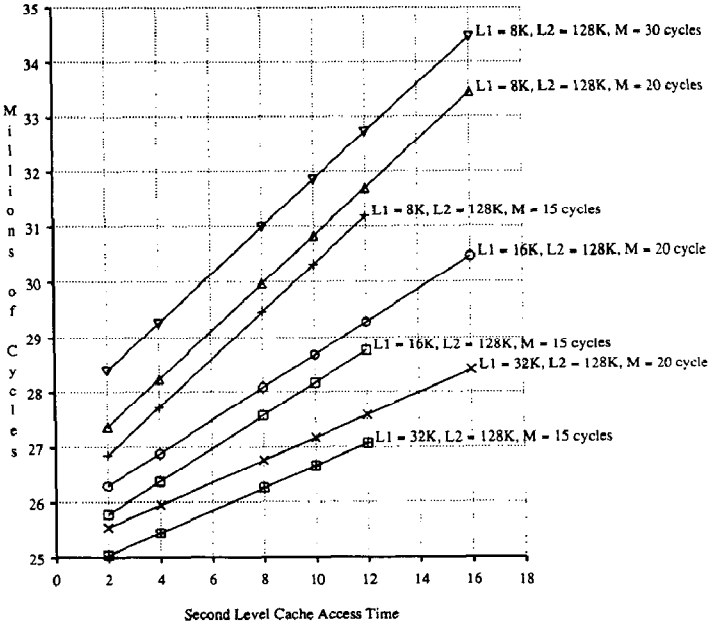


Figure 7: Execution Time vs. L2 Cache Access Time

As previously stated, high on-chip densities make it possible to include small on-chip caches on current microprocessors. It is interesting to know whether even tiny on-chip caches are worthwhile, particularly when backed up by second-level caches. We modelled small L1 caches with sizes varying from 128 bytes to 4096 bytes, backed up by L2 caches of 8K to 128K bytes, to check the utility of such configurations. These measurements assume a 1-cycle L1 cache and a 4-cycle L2 cache. For comparison, we also show the performance of a single 1-cycle L1 cache and a single 4-cycle "L2" cache.

The results are summarized in Figure 8. First, we see the significant impact of adding a second level; adding an 8K L2 cache to the 128 byte L1 cache nearly doubles the performance. Adding an on-chip cache is also significant; compared with no L1 cache, a 128-byte L1 cache gives a 35 percent performance over a single 8K L2 cache. Given a two-level structure, the size of the L1 cache is clearly the most crucial parameter. With the 128 byte L1 cache, changing the size of the L2 cache from 8K to 128K produces less than a 12 percent improvement. This is because the four cycles needed to process L1 cache misses are large enough that eliminating misses from the second level is relatively insignificant. The effect is similar on larger L1 caches but is magnified by the higher miss ratio in small local caches.

3.3 Write Strategy

Different write strategies were modelled by our simulation. As might be expected, the use of a write-back strategy can substantially reduce traffic to the next level in a multi-level hierarchy.

In the two-level cache, the choice of write strategy can be made independently at each level. Figure 9 shows the performance of our simulated system with 8K L1 cache using various write strategies. In this case, if write-back is to be used at only one level, it should be used in the L1 cache, since this will eliminate most of the write traffic between the two caches. Unfortunately, we could not measure the effect of I/O interference on an L1 cache with write-back. On disk writes, for example, the L2 cache would not have up-to-date copies of recently written data, so I/O would have to pass through the L1 cache or the L1 data would have to be written back before the I/O begins. Disk reads also present consistency problems.

A write-back L2 cache with write-through L1 cache provides about 5 percent lower performance than write-through L2 with write-back L1. This is the result of the increased inter-cache write traffic. Because the cycle time of the L2 cache is not as critical as that of the first level, it may be more practical to include the write-back strategy in L2 than in L1. If the L1 cache is write-through and multi-level inclusion [4] is imposed (i.e., L1 is a subset of L2), then only the L2 cache directory needs to be checked on I/O operations; still, invalidations will need to be passed up to the L1 cache.

As shown in Figure 9, the best performance is achieved by using a write-back strategy in both caches. However, this also requires the most complex implementation and the small increase in performance is probably not worthwhile.

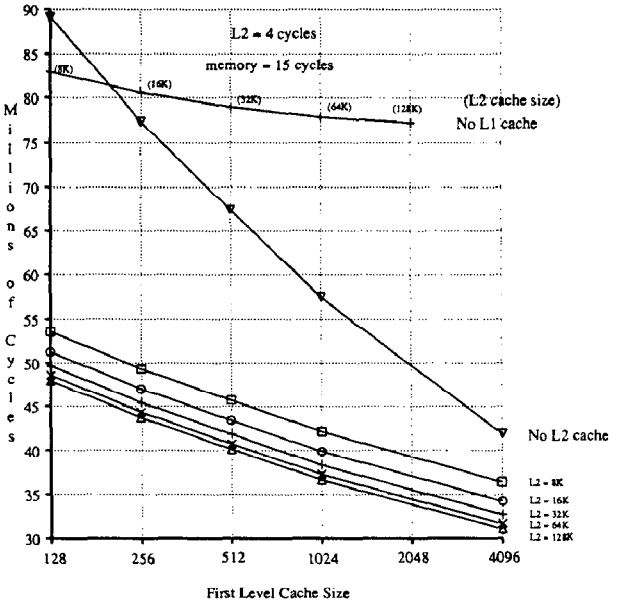


Figure 8: Performance of Small L1 Caches

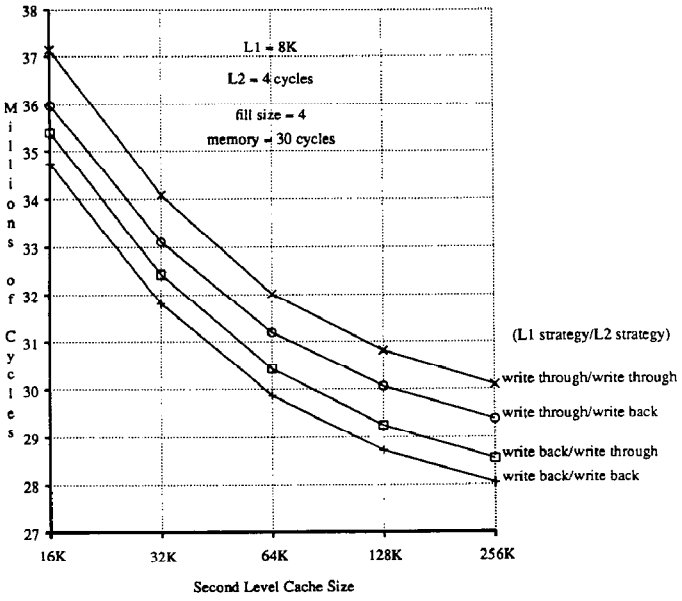


Figure 9: Effect of Write Strategy

3.4 Sequential Cache Misses

In a high speed system it may be possible to process more than one memory request simultaneously, i.e., when a read misses in the cache and a request for the data is sent to memory the next sequential read can be started before the first has completed. With a long memory access time and several independent memory banks, the first reference will require the entire memory access time to complete. But, if a following reference can be started before the first finishes, there will be a significant performance advantage. If one provides this feature in the cache controller the next question to be addressed is how many memory references should be processed at a time?

Our simulation is capable of creating a histogram of the number of adjacent misses in the local cache. The results show that smaller caches with low hit ratios would benefit greatly from the ability to be able to process two reads simultaneously (10% - 15% fewer cycles). The incremental improvement made possible by increasing the number of simultaneous reads to three is extremely small, on the order of one percent. However, as the cache size is increased there is a much smaller performance improvement. With a 32Kbyte local cache and a 30 cycle memory access time the improvement is approximately five percent. The hardware to control two outstanding references does not seem very complicated and would appear to be justified.

3.5 Separate Instruction and Data Caches

The effect of separating the instruction and data caches and of varying the size and fill rate for each was modelled. With our simulation, we found little performance difference between a separate I and D cache and a shared cache with the same total cache size and access time. Separating the caches provided a performance increase in the range of five to ten percent. The extra complexity of adding a complete set of cache control logic and an extra cache directory would not seem to be cost effective.

This result is an effect of the complex instruction set architecture. In our simulated architecture the average instruction takes 7 cycles; a cache access takes 1 cycle, so there is plenty of time during an instruction to retrieve multiple items from the shared cache. In a RISC-like architecture with a small number of cycles per instruction, separate I and D caches may be required because the I fetches will saturate the cache.

On the other hand, separate I and D caches may have an advantage in our system. As previously stated, larger memories typically have longer access times due to physical constraints; the shared cache model, with the same total amount of memory, might require a slower clock frequency than would separate caches. In effect, adding an extra cache may allow us to double the effective size of the cache without decreasing the system cycle time. This performance increase would be in the range of ten to almost twenty percent and may justify the extra expense and complexity.

4 Conclusions

We have studied several aspects of two-level cache memories in uniprocessors. In particular, we examined the hit ratio and performance impact of varying the sizes of first- and second-level caches. We also simulated the effect of splitting the local cache into separate data and instruction parts.

An extra level of cache memory can provide a worthwhile performance gain when used with proper combinations of small first-level caches and large main memory access times. Adding a small amount of fast first-level cache (e.g., a few hundred words on a VLSI chip) to a system with a four- to six-cycle second-level cache can also achieve substantial gains. However, in some cases with relatively fast main memories or slow second-level caches, the addition of the second cache can actually degrade performance. Starting the L2 lookup and the primary memory access in parallel will alleviate this problem but at the cost of wasted bus and memory bandwidth.

For our model and data, separate caches for instructions and data did not by themselves produce a worthwhile performance increase. However, because smaller memories have shorter access times providing separate instruction and data caches allows more memory to be used without a decrease in the processor cycle time. Separate instruction and data caches may be more worthwhile in a system with fewer cycles per instruction.

Write-through and write-back strategies were studied for both cache levels. The best performance is provided by write-back at both levels, however this method is also the most complex to implement. The principal effect of write-back was the reduction of write bandwidth required at the next higher level of memory. If only one of the caches were to use write-back, then write-back should be included in the L1 cache because (1) the L1 cache typically has a faster access time, and writes could be processed more quickly without buffering, and (2) the L1 cache is smaller than the L2 cache. Also , a write-back cache must include error correction logic for reliability; thus, there may be significant cost saving in using write-back at L1 and write-through at L2.

However, there are other issues that might affect this decision. First, a write-back cache is more complex and might lengthen the L1 access time when compared to a write-through cache. Second, if the L1 cache is write-back then the two caches

will be inconsistent; this may cause problems for I/O operations. Therefore, a write-through L1 and write-back L2 might make more sense for engineering reasons; it is simpler to implement and still reduces traffic between the L2 cache and main memory.

Multilevel caches provide a fruitful area for study, and much more needs to be done. The number of relevant parameters that can be varied is large and the amount of data that can be collected and analyzed is tremendous. For this paper, we have fixed many of the parameters to show representative examples of our results. Obviously, changes to these parameters would cause different results; for example, a larger line size may either increase or decrease performance, and different L1 and L2 line sizes will certainly be used in actual implementations.

Certainly an important area for further evaluation is multilevel caches in multiprocessors. In this study, we concentrated on uniprocessors, partially because of the difficulty of obtaining multiprocessor traces.

5 Acknowledgements

We would like to thank Jean-Loup Baer, Jon Bertoni, Wen-Hann Wang, Sang Min, Haim Mizrahi, and Jorgen Staunstrup for their reviews of early drafts of this paper.

References

[1] Anant Agarwal, Paul Chow, Mark Horowitz, John Acken, Arturo Saltz, and John Hennessy. On-chip instruction caches for high performance processors. In *Conference on Advanced Research in VLSI*, March 1987.

[2] Anant Agarwal, Richard L. Sites, and Mark Horowitz. ATUM: a new technique for capturing address traces using microcode. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 119–127, ACM SIGARCH, June 1986.

[3] David Archer, David Deverell, Thomas Fox, Paul Gronowski, Anil Jain, Michael Leary, Daniel Miner, Andrew Olesin, Shawn Persels, Paul Rubinfeld, and Robert Supnik. A CMOS VAX microprocessor with on-chip cache and memory management. *IEEE Journal of Solid State Circuits*, SC-22(5):849–852, October 1987.

[4] Jean-Loup Baer and Wen-Hann Wang. Architectural choices for multi-level cache hierarchies. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 258–261, August 1987. Also (in an expanded form) Department of Computer Science Report TR-87-01-04, University of Washington, January 1987.

[5] Fayé Briggs and Michel Dubois. Effectiveness of private caches in multiprocessor systems with parallel-pipelined memories. *IEEE Transactions on Computers*, C-32(1):48–59, January 1983.

[6] Paul Chow and Mark Horowitz. Architectural tradeoffs in the design of MIPS-X. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 300–308, June 1987.

[7] Douglas W. Clark. Cache performance in the VAX-11/780. *ACM Transactions on Computer Systems*, 1(1):24–37, February 1983.

[8] Douglas W. Clark. Pipelining and performance in the VAX 8800 processor. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 173–177, October 1987.

[9] Douglas W. Clark and Joel S. Emer. Performance of the VAX-11/780 translation buffer: simulation and measurement. *ACM Transactions on Computer Systems*, 3(1):31–62, February 1985.

[10] Daniel J. Colglazier. *A Performance Analysis of Multiprocessors Using Two-Level Caches.* Master's thesis, University of Illinois, 1984.

[11] David R. Ditzel, Hubert R. McLellan, and Alan D. Berenbaum. The hardware architecture of the CRISP microprocessor. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 309–319, June 1987.

[12] R. P. Fletcher, R. A. Heller, and D. M. Stein. MP–shared cache with store-through local caches. *IBM Technical Disclosure Bulletin*, 25(10):5133–5135, March 1983.

[13] James R. Goodman. Using cache memory to reduce processor memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, June 1983.

[14] Akira Hattori, Minoru Koshino, and Shigemi Kamimoto. Three-level hierarchical storage system for the FACOM M-380/382. In *Proceedings Information Processing IFIP*, pages 693–697, 1983.

[15] Mark Hill, et al. Design decisions in SPUR. *Computer*, 8–22, November 1986.

[16] Henry M. Levy and Peter H. Lipman. Virtual memory management in the VAX/VMS operating system. *Computer*, 35–41, March 1982.

[17] John S. Liptay. Structural aspects of the System/360 Model 85, Part II – The cache. *IBM Systems Journal*, 7(1):15–21, 1968.

[18] Robert T. Short. *A Study of Multilevel Cache Memories.* Master's thesis, Department of Computer Science, University of Washington, January 1987.

[19] Alan J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[20] Alan J. Smith. Line (block) size choices for CPU cache memories. *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

[21] Alan J. Smith. Problems, directions and issues in memory hierarchies. In *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, pages 468–476, 1985.

[22] F. J. Sparacio. Data processing system with second level cache. *IBM Technical Disclosure Bulletin*, 21(6):2468–2469, November 1978.

[23] William D. Strecker. Cache memories for PDP-11 family computers. In *Proceedings of the 3rd Annual Symposium on Computer Architecture*, pages 155–158, January 1976.

[24] Andrew W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 244–252, June 1987.

[25] Phil C. C. Yeh, Janek H. Patel, and Edward S. Davidson. Shared cache for multiple-stream computer systems. *IEEE Transactions on Computers*, C-32(1):38–47, January 1983.