Performance Tradeoffs in Cache Design

Steven Przybylski, Mark Horowitz, John Hennessy

**Computer Systems Laboratory, Stanford University,
Stanford University, CA., 94305**

Abstract

Cache memories have become common across a wide range of computer implementations. To date, most analyses of cache performance have concentrated on time independent metrics, such as miss rate and traffic ratio. This paper presents a series of simulations that explore the interactions between various organizational decisions and program execution time. We investigate the tradeoffs between cache size and CPU/Cache cycle time, set associativity and cycle time, and between block size and main memory speed. The results indicate that neither cycle time nor cache size dominates the other across the entire design space. For common implementation technologies, performance is maximized when the size is increased to the 32KB to 128KB range with modest penalties to the cycle time. If set associativity impacts the cycle time by more than a few nanoseconds, it increases overall execution time. Since the block size and memory transfer rate combine to affect the cache miss penalty, the optimum block size is substantially smaller than that which minimizes the miss rate. Finally, the interdependence between optimal cache configuration and the main memory speed necessitates multi-level cache hierarchies for high performance uniprocessors.

1. Introduction

Caches are currently found in all classes of computers from personal computers to supercomputers. Their prominence has been the result of their increasing benefit and decreasing cost. Technological improvements of the past decade have dramatically improved the processor performance with respect to main memory speed. As processor cycle times fall, an increasing proportion of the peak processing power is lost to the memory system. These same technological advances have decreased the relative cost of adding a small cache, both on and off-chip. Caches are, and will remain, one of the simplest and most effective way of improving execution time.

It is not surprizing then that cache design has been extensively studied [16]. Each of the major design decisions has been the focus of many papers. With some notable exceptions [13, 8], the work to date has ignored in two respects the important dimension of time. First, the focus of the previous efforts has been on the effects of cache organization on miss rates and traffic ratios. These metrics of cache performance remain unchanged if the cache miss penalty is 2 cycles or 20, and they do not accurately reflect the processor's performance. The goals and constraints that computer designers deal with are those of peak and sustained performance, cost, area and power; not bus utilization and miss rates. Though these popular cache metrics are related to total execution time in apparently simple ways, they can, as we shall see, be deceptive.

The second way in which time is neglected is that the various organizational decisions (set size, number of sets, block size, etc) directly or indirectly affect the cycle time of the cache. Larger caches require more or larger RAMs. A larger array increases loading on critical address and data lines, requiring additional buffering. Larger and wider RAMs are typically slower and more expensive. Set associativity may require an additional multiplexor in the data path, as well as affecting the complexity and timing of the control. These interactions are often non-linear, subtle, and unique to a particular design.

Execution time is the product of number of cycles executed and the cycle time. Concentrating on miss ratios de-emphasizes organizational tradeoffs that affect the cycle count. Ignoring the temporal implications of cache characteristics discounts entirely tradeoffs that involve the system cycle time. Faced with the need to reconcile performance goals with organizational and physical constraints, machine designers have always been forced to deal with both classes of tradeoffs. However, since computer designers are always focused on a particular implementation, they tend to study only the local optimizations around a particular design point. The results presented here are part of a larger effort to examine these interdependencies across a significant portion of the design space, and to focus cache design on overall system performance.

In reality, the temporal tradeoffs are complicated by the link between the cache cycle time and the CPU cycle time. A change in the cache's organization may or may not affect the processors cycle time, depending on the critical paths in the design. To eliminate this ambiguity, we uniformly assume that the system cycle time is determined by the cache.

The primary tool in this study is trace driven simulation. To contain the explosion in the design space that results from the addition of temporal variables, we focus on high performance engineering workstations of the near future. This partial spanning of the entire design space limits wide-scale applicability of the numerical results presented here. However, the simulations provide insight into the underlying mechanisms and interactions, which operate over larger fraction of the entire design space.

The next section of this paper describes the simulator and analysis environment, as well as the traces used as stimulus. The simulator was specifically designed to efficiently model a wide range of computer systems. Section 3 looks at the primary tradeoff between CPU cycle time and cache size. It strongly indicates that pushing cache size to extremes to benefit either cycle time or miss rate is not warranted. Section 4 looks at the performance impact of set associativity across a wide range of systems, and concludes that for systems built of TTL components, set associativity is unlikely to improve performance. Section 5 looks at the dependence of cache design on main memory design. These two components have traditionally been considered independently, but in reality the main memory characteristics affect the optimal cache configuration. Specifically, this section deals with the choice of block size. Section 6 considers the implications of the earlier results on cache organizations and computer system design as a whole. It makes a strong argument for multilevel cache hierarchies as the best way to get the most out of high performance CPUs. Section 7 summarizes and concludes the paper.

2. Simulation Environment

The primary requirement to investigate temporal tradeoffs in cache design is a simulator that has a comprehensive model of time, and which can generate accurate and credible cycle counts. In addition to all the usual organizational parameters (total size, set size[1], block size, fetch size[2], write strategy, write buffering), the user can vary the number of machine cycles that reads and writes take at each each level of the memory hierarchy. Not only is the duration specified, but the separation between operations, and any communication delays between layers can be set as well. There is also provision for synchronization costs that occur if different function units run at different rates. The CPU modeled in the simulator is a pipelined machine capable of issuing simultaneous instruction and data references. If there are separate instruction and data caches then, instruction and data references in the trace paired up without reordering any of the references. These couplets are issued at the same time and both must complete before the CPU can proceed to the next reference or reference pair. Write buffers are included between every level of the modeled system. With eight parameters, the write buffer model can replicate any reasonable write strategy. The write buffers check the addresses of reads to make sure that the fetched data is not stale. In the case of a match, the read is delayed until the write propagates out of the buffer and into the next level of the hierarchy. Virtual to physical translation can be placed anywhere in the hierarchy. All the simulations presented here are with virtual caches, which include the process identifier with the high order address bits in the tag field.

A simulation run consists three phases: macro expansion, compilation and simulation. The macro expansion phase begins with pointers to a system specification file and two or three variation files. The specification file dictates which of two cache models is to be used at each level of a multi-level cache hierarchy and specifies the default value of all the parameters. About 130 parameters are needed to fully specify a two level cache system. Each of the variation files changes one or more characteristics: for example, set size, number of sets, cycle time, or memory latency. A change could involve several parameters in order to maintain consistency in the modeled system. The result of the first phase is a C program in which all the parameter expressions have transformed into constants. Furthermore, all tests based on system parameters are resolved by the macro preprocessor, so that there is no run-time overhead for the flexibility in the system. For very long traces, the improvement in simulation rate overcomes the penalty associated with the the first two phases. To further reduce the run-time effort, each trace is preprocessed to extract all the system independent statistics. During simulation, up to about 400 unique statistics are gathered, to produce an 18KB raw data file. For each tradeoff studied, a custom program reads in the raw data files and generates the graphs and tables presented in this paper. By farming out a series of simulations to between 10 and 20 MicroVax II workstations, aggregate simulation rates of 38,000 references per second are obtained. This rate is achieved without compromising the range of systems that can be simulated, or the variety of statistics gathered.

Despite these simulation rates, it is only possible to properly explore a limited region of the overall cache design space. The region examined in this paper is centered around one particular Harvard organization. The split I and D caches are 64 kilobytes (KB) each, organized as 4K blocks of four words[3] (W), direct mapped. Entire blocks are fetched on a miss. The data cache is write back, with no fetch done on write miss. All read hits take one CPU cycle, while writes take two – one to access the tags, followed by one to write the data. A four block write buffer is provided for. It is of sufficient depth that it essentially never fills up. In the case a dirty miss, the memory read is started immediately, and the dirty block is transferred into the write buffer during the memory latency period. If the latency is sufficiently long, then the write back is completely hidden. However, since all the data paths are set to be one word (W) wide, this is not always the case for long block sizes. The base CPU cycle time is 40ns and the CPU and cache cycle times are assumed to be identical.

Main memory is modeled as a single functional unit. Read access times are made up of a latency portion, followed by a transfer period. The default latency is one cycle to get the block address to the memory, plus 180ns. Since the memory is synchronous, the latency becomes $1 + \lceil 180ns/40ns \rceil$, or 6 cycles. The transfer rate is one word per cycle, or four cycles for a block. Based on the difference between DRAM access and cycle times, once a read is completed, at least 120ns (3 cycles in the default case) must elapse before the start of the next operation. Writes take one cycle for the address, followed by one word per cycle transfer. At this point the cache can proceed with other business while the write actually occurs, which takes 100ns followed by the same 120ns recovery time. On write backs, the entire block is transferred, regardless of which words were dirty.

These parameters were chosen to be representative of a machine built around a 25 MIPS peak CMOS RISC processor. In particular, the memory system is quite aggressive by today's standards. The backplane has more than double the transfer rate of VME or MULTIBUS II, and memory latency is roughly a half that of commercially available boards for these busses. The values used are more representative of a single master private memory bus.

The traces used as input to the simulator are listed in Table 1. The first four are catenations of several 400,000 reference snapshots taken on a VAX 8200 running with microcode modifications to catch generated virtual addresses [1]. They include operating system references and exhibit real multiprogramming behaviour. Rather than deal with the variety of VAX data types, the traces have been preprocessed to contain only word references. Specifically, sequences of instruction fetches from the same word are collapsed into a single word reference, and quad references were split into several sequential accesses. The warm start boundary for these was consistently 450,000 references. Though it is unrealistic to expect an implementation of the VAX architecture to issue memory references at the rate of the CPU model, the traces are valuable because of the multiprogramming and operating system characteristics. To use a less demanding CPU model for these traces of traces would unnecessarily complicate the interpretation of the results.

Name	Processes	Size (K Refs)	Unique Addresses (K)	OS	Programs
mu3	7	1439	33.1	VMS	Fortran compile, microcode allocator, directory search.
mu6	11	1543	49.6	VMS	mu3 + Pascal compile, 4x1x5, spice.
mu10	14	1094	49.4	VMS	mu6 + jacobian, string search, assembler, octal dump, linker.
savec	6	1162	25.2	Ultrix	C compile with miscellaneous other activity.
rd1n3	3	1489	299	–	emacs, switch ,rsim C compiler front end (ccom).
rd2n4	4	1314	241	–	emacs, troff, a trace analyzer program, ccom.
rd1n5	5	1314	248	–	rd2n4 + egrep searching 400KB in 27 files.
rd2n7	7	1678	448	–	rd2n4 + rsim, grep doing a constant search, emacs.

Table 1: Description of the Traces

The second four traces are interleaved uniprocess, virtual address traces of the MIPS Computer System's R2000 processor, running optimized C programs. Though no operating system references are present, the programs are randomly interleaved to duplicate the distribution of context switch intervals seen in the VAX traces. The traces are gathered from a random location in the middle of each program's execution so that steady state behaviour is represented. The grep and egrep programs were observed from the start of execution to characterize program start-up. The first portion of each uniprocess trace

[1]Set size is used synonymously with degree of associativity. The number of sets is $2^{\text{number of set bits}}$, where the set bits are the portion of the address used to index into the cache.

[2]The fetch size is called the transfer size by Smith [11].

[3]A word is defined to be 32 bits.

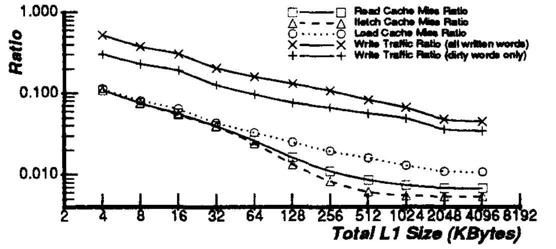
contains all the unique references touched by the programs up to the time at which tracing was begun. These references are in the order of their most recent use and interleaved at the start of the aggregate traces with the same distribution. By prepending these references, the cache contents at the warm start boundary is very similar to what it would be if the programs where simulated from their beginning. This is true regardless of the cache organization. These initialization references account for the larger number of unique references in the R2000 traces. Traces generated in this way have the advantage that simulation results based on them are valid for very large caches. For these four traces, data was gathered over the last one million references.

Though various specific differences between the two groups of traces are addressed later, they behaved similarly both qualitatively and quantitatively in all the experiments presented here. Numerical results in this paper are the geometric mean of warm start runs for all eight traces.

3. Speed – Size Tradeoffs

The tradeoff between cache size and cycle time is usually phrased in terms of maximizing the cache size without increasing the given CPU cycle time. This however is a biased perspective, in that it treats as secondary the large impact that the cache has on overall performance. It is much more appropriate to choose a cycle time that accommodates the needs of both the CPU and cache in such a way that optimizes performance.

The usual cache metrics are miss rates and transfer ratios. Figure 3-1 confirms the widely held belief that larger caches are better, but that beyond a certain size, the incremental improvements are small [2, 6, 12]. 'Total L1 Size' refers to the sum of the data portions of the two caches at the first (and only) layer of caching. The two caches were varied together from 2KB though 2MB, so that the total cache size ranges from 4KB to 4MB. The block size and other cache parameters were kept constant as the number of sets was increased. The miss ratios are calculated as read[4] misses per read requests, as opposed to being relative to the total number of references. The RISC traces showed consistently lower miss rates. For small and medium sized caches, the load miss rate was between 11.5% and 18% lower. The instruction miss rate was between 29% and 46% lower, reflecting a higher degree of locality and a lower instruction density. For large caches, the accurate cache initialization in the R2000 traces further reduces the instruction miss rates compared with the VAX traces. These factors account for miss rates that are somewhat lower than Smith's Design Target Miss Ratios [13].

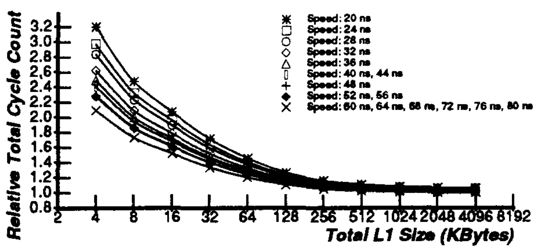


Cache Metrics

Figure 3-1

The larger of the two write traffic ratios counts all words in blocks that are dirty when replaced, while the smaller counts only the dirty words themselves. A small proportion of words in a block that are dirty on write back yields a large separation between the two write traffic ratios. In the system modeled, no fetching occurs on a write miss, so the write miss ratio is not interesting. Higher write transfer rates with for RISC traces at large cache sizes result from the zeroing of the data space at the start of the grep and egrep processes. Since the block size is fixed, and all references are assumed to be word references, the read traffic ratio is simply four times the miss ratio.

The other parameter involved in the speed – size tradeoff is the cycle time. As the CPU/cache cycle time is varied over the range of 20ns through 80ns, the total cycle count for the traces decreases, giving the illusion of improved performance. In Figure 3-2, the cycle counts are normalized to the smallest count in the experiment, that being for two 2MB caches operating at 80ns. The decrease in cycle count with increasing cycle time stems from the reduced number of cycles per memory reference at the slower clock rates. The total number of cycles to execute a program varies quite dramatically – by a factor of 3.2 across the entire experiment, and by a factor of 1.5 when each cache is 2KB. As the cache size increases the number of misses drops, causing the main memory component of the total cycle count to decrease as well. Table 2 shows that the cost in cycles of each type of operation changes with the cycle time, since the latency portion takes a constant amount of time.



Speed - Size Tradeoff: Cycle Count

Figure 3-2

Cycle Time (ns)	Read Time (cycles)	Write Time (cycles)	Recovery time (cycles)
20	14	10	6
24	13	10	5
28	12	9	5
32	11	9	4
36	10	8	4
40	10	8	3
48	9	8	3
52	9	7	3
60	8	7	2

Read Operation Time: 180 ns
Write Operation Time: 100 ns
MM Recover Time: 120 ns

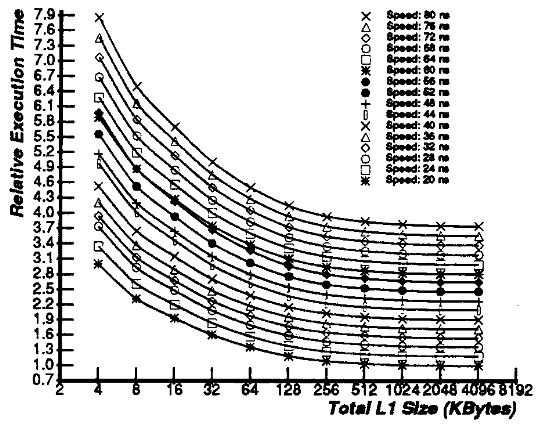
Table 2: Memory Access Cycle Counts

Total execution time is the product of cycle time and cycle count. Figure 3-3 shows that the overall performance is strongly dependent on both the cache size and cycle time. With small caches, incremental changes in the cache size have a greater effect than changes in the cycle time, while at the larger cache sizes the reverse is true. This strongly indicates that there is a likely to be a region in the middle where both are appropriately balanced.

As an aside, an unusual phenomenon occurs for systems in the vicinity of 60ns with small cache sizes. Strangely enough, performance is not monotonic function of cycle time. Decreasing the cycle time from 60ns to 56ns slows the machine down close to 3%. Since the memory access time is quantized to a discrete number of cycles, the cache miss penalty increases from 8 to 9 cycles at this boundary. Because of the poor miss rate for small caches, the total cycle count increase that results easily overwhelms the decrease in cycle time to increase the total execution time. The 56ns design is a poor one in that a significant fraction of the read access time is being wasted in synchronizing the

[4]A read is defined to be either a load or an instruction fetch (ifetch).

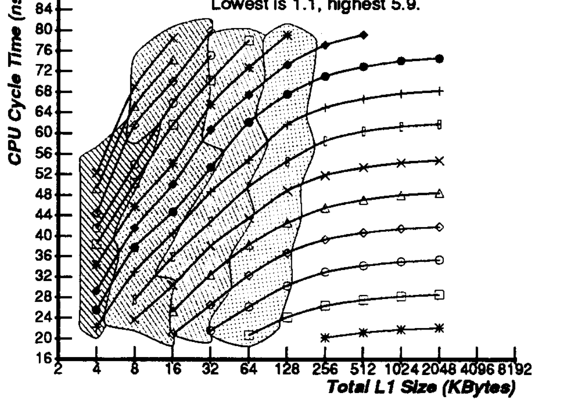
asynchronous DRAM access with the synchronous backplane and cache.



Speed - Size Tradeoff: Execution Time

Figure 3-3

Horizontal slices though Figure 3-3 expose groups of machines with equal performance. By vertically interpolating between the simulations of the same cache size, we can estimate the cycle time required in conjunction with each cache organization to attain any given performance level. This interpolation process smoothes the quantization effects to the point where they are inconsequential. The results of the interpolation, shown in Figure 3-4, follow lines of equal performance across the design space. The best performance level displayed is 1.1 times slower than the (4MB, 20ns) scenario. The increment between the lines is an increase in execution time equal to 0.3 times this normalization value. The uppermost line depicts a class of machines that run 5.7 times slower[5] then the best case for this experiment.



Lines of Equal Performance

Figure 3-4

The slope of the lines indicates how much cycle time can be exchanged for a doubling in cache size without changing the overall performance. A slope of 5ns per doubling in size means that a straight swap of cache RAMs for ones four times as big (typically the next size up) but 10ns slower is performance neutral. For this slope, less than a 10ns degradation in cycle time improves performance, while more than 10ns of additional cycle time moves the machine to a design point that is on a poorer performance curve. The number of lines crossed in going from one design point to another indicates the difference in execution time between the two machines. The shaded areas are regions of the design space in which the slope of the curves fall into specific ranges. In the unshaded region at the right, the slopes are less than 2.5ns per doubling, while in the leftmost shaded zone the slopes are consistently greater than 10ns per doubling. The four boundaries between the five regions are the 2.5ns, 5ns, 7.5ns and 10ns per doubling contours.

The four shaded regions are the portions of the design space in which an exchange of discrete static RAMs for the next size up is beneficial if the speed difference is 5ns or less through 20ns or less, the latter being the leftmost. Conversely, the delineated zones are the portions the design space in which it is beneficial to exchange the RAMs for smaller yet faster ones. In the unshaded area, such an exchange is warranted if the cycle time of the machine drops by only 5ns, again assuming a factor of four change in cache size. With each region successively to the left, the cycle time improvement needed to justify the swap increases by 5ns.

For example, consider a system being built around a 40ns CPU, requiring 15ns RAMs to attain that cycle time. If the best available 16Kb and 64Kb RAMs run at 15 and 25ns respectively, then two comparable design alternatives are 8KB per cache with the 2K by 8b chips or 32KB per cache with the 8K by 8b chips. Both contain the same number of chips in the same configuration. The only changes between them are the number of tag bits, the width of the cache index and the system clock frequency. The slope of constant performance curve at the (16KB,40ns) design point is 16ns per quadrupling, greater than the 10ns difference in the RAM speeds. As a result running the CPU at 50ns with a larger cache improves the overall performance by 7.3%[6].

It's important to realize that in this case, performance is maximized when the CPU is running at less than its maximum frequency. This makes sense from the perspective of mean time per reference – a critical parameter for all architectures, but especially RISC machines which generally rely on an effective memory hierarchy. If a small cache has a miss rate of 10% and a 10 cycle miss penalty, then the average time per read is 2 cycles. A reduction in the miss rate to 9% reduces the average time per reference to 1.9 cycles. Thus the cycle time can degrade by 5% without affecting the cost of making memory references.

Since memory access time has a major component that is independent of cycle time, the miss penalty increases as the cycle time decreases. As the miss penalty increases, a fixed change in the miss rate has a greater impact on performance. Alternatively, as the miss penalty increases, a fixed change in the miss rate is equivalent to an increasing proportion of the cycle time. The combined effects of a decreasing cycle time and an increasing fraction of a cycle that is equivalent to a change in cache size cause the cycle time to miss rate tradeoff to be relatively constant. This explains why the regions of Figure 3-4 are nearly vertical: the cycle time – cache size tradeoff is independent of the cycle time.

If a less aggressive memory model was used in the simulated system, the smaller caches would have still worse performance relative to the larger caches. The higher miss rates increase the sensitivity of systems with smaller caches to the memory access time. This would have the effect of bending the lines of constant performance down for small cache sizes, and increasing their slope across the entire range of sizes. This in turn increases the desirability of larger caches at the expense of cycle time.

An inevitable consequence of the asymptotic shape of the lines of constant performance is that there is a strong tendency to increase cache size to the 32KB to 128KB range. Beyond that region, the curves are sufficiently flat that there is little motivation for further increases in cache size, regardless of the minuteness of the cycle time penalty. Once that design goal is reached, any additional hardware and money is most effectively spent improving the cycle time of the cache/CPU pair.

[5]Being the 16th line from the bottom, its performance level is $1.1 + (16-1) \times 0.3 = 5.9$.

[6]With the smaller cache, the execution time is 3.15 times the best case for the experiment, while with the 64KB of cache, the mean execution time is only 2.92 times the normalization point, yielding a 7.3% improvement.

4. Set Associativity

This section examines the costs and benefits of set associativity. As in the previous section, the miss rate improvement accompanying an increase in set associativity is equated to a change to the cycle time of the processor/cache pair.

The motivation for set associativity is two fold. First of all, there is a reduction in the miss rate over a direct mapped cache of the same size [15]. Frequently, a large set size is indirectly imposed by the virtual memory constraints. Unrestricted aliasing of multiple virtual pages to a single physical page is most effectively handled by doing the virtual to physical translation before or in conjunction with the cache access. If performance requirements require simultaneous access of the cache and address translation hardware, then only the virtual address is available when the access of the physically addressed cache is begun. If more than the page offset bits[7] are used in the cache access, the data may reside in any of several sets [18]. If this is to be avoided, then the only way to increase the cache size is to increase the set size. For example, the IBM 3033 has a 16 way set associative 64KB cache [14] for this reason. Restricting the virtual to physical map [10] and increasing the page size postpone this problem by increasing the maximum number of sets. It is possible to allow multiple cache sets to potentially hold a piece of data, at the expense of control complexity and degradation in the mean cache hit time [5].

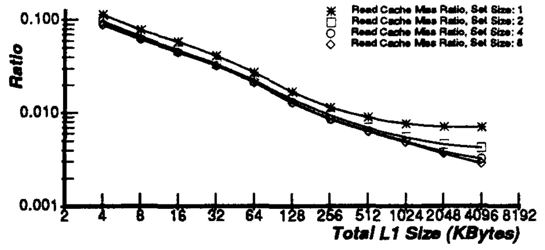
The cost of set associativity is often difficult to assess in absolute terms. Though the number of tags is not affected by the associativity, data path widths are directly related to the set size. If the cache is distributed across several integrated circuits, increasing the number of bits that must be read simultaneously can dramatically affect the area and cost of the cache.

The biggest temporal impact of set associativity is the need to complete the tag access and comparison before gating the data to the CPU. Whether this gating is done via explicit multiplexors, or via output enables onto a tri-state bus, there is likely to be an incremental delay to the cache's access time over a direct mapped implementation. Depending on the physical constraints, the additional access width may mandate broader static RAMs. Within a single product family, wider yet shallower RAMs are invariably slower than the by-one and by-four versions. A wide physical organization may result in additional loading on critical address and data lines, causing further degradation in the cycle time. The accumulated time penalty is even more severe when the cache/CPU interface would otherwise allow the hit/miss information to arrive after the data [9]. There is no opportunity to use this additional time since the choice of which piece of data to present to the CPU is dependent on which set, if any, matched.

The cycle time penalty of an increased set size can be lessened by using faster RAM technology for the tags. This allows for overlap of the tag comparison with a portion of the data access. Depending on the block size and the physical organization of the cache, the data portion can be made deeper than the tag portion.

All of these degradations are most severe for large caches made of discrete RAMs. For integrated caches, they are all substantially reduced. Since the performance of a RAM array degrades as it becomes less square [4], any moderately large array fetches several words simultaneously. The multiplexors needed in a set associative design would be necessary regardless of the cache organization: the challenge is to complete the tag fetches and comparisons sufficiently early that the control signals to do not delay the data. In a full custom VLSI design, the degradation due to set associativity can be minimized through the use of physical and logical organizational optimizations and circuit techniques.

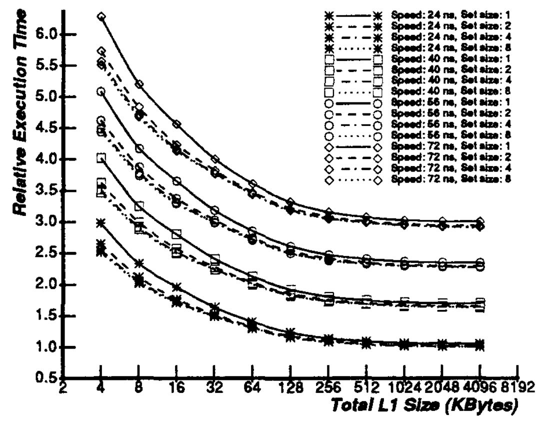
Rather than try to quantify these various temporal and physical costs, we have translated the benefits associated with the improved miss ratio into equivalent cycle time changes. If the implementation of set associativity impacts the cache/CPU cycle time by an amount greater than this break-even value, then adding set associativity is detrimental to overall performance.



Read Miss Ratio

Figure 4-1

Figure 4-1 essentially confirms previously reported results for the effects of set associativity on miss rate [8, 3]. Again, the X axis plots the combined size of the equal I and D caches. As the total cache size is being kept constant, a doubling in associativity is accompanied by a halving of the number of sets. Random replacement is used regardless of the set size. The change from direct mapped to two way set associativity drops the miss ratio[8] by about 20% for caches up to about 256KB total. Above that the improvements increases because the caches are virtual. Above this level, neither intra- nor inter-process conflicts are eliminated by adding more sets, while both, and in particular the latter, are significantly helped by increasing the associativity. Smaller improvements are seen for set sizes above two.



Speed - Size Tradeoff: Execution Time

Figure 4-2

As in the previous section, the tradeoff between cycle time and set associativity is investigated by varying both parameters simultaneously. Figure 4-2 shows the effect on execution time of cache size, set associativity and cycle time. The memory model is the same as used previously, with the cycles per memory reference varying according to Table 2. Comparing with Figure 3-3, a change in associativity can be seen to have a significant performance effect for the smaller caches. A 10% improvement in execution time is seen for a 4KB combined cache size for a change in set size from one to two. For large caches, the improvement is much less significant. Since the miss rate is smaller for larger caches, the main memory accounts for a smaller proportion of total execution time. A constant percentage decrease in the number of

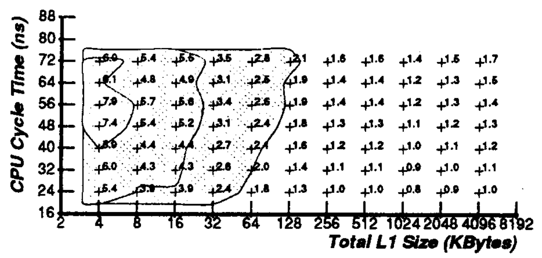
[7]The page offset and block offset bits are those that are unaffected by the translation.

[8]Called the miss ratio spread by Hill [8]

memory accesses corresponds to a smaller percentage decrease in total execution time.

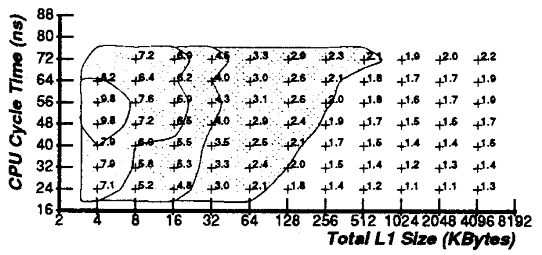
Vertical interpolation between solid lines allows estimation of the cycle time that a direct mapped machine would need to match the performance of a set associative design of the same size. The difference between the cycle times of the two machines is the amount of time available for the implementation of set associativity. A degradation in cycle time greater than this difference results in a net decrease in performance[9].

Figures 4-3 to 4-5 map the break-even degradations in cycle time for set associativities of two, four and eight across the explored design space. The boundaries between the shaded regions are the contours at 2ns through 10ns at 2ns intervals. They show the portions of the design space in which more or less time is available for the break-even implementation of associativity.



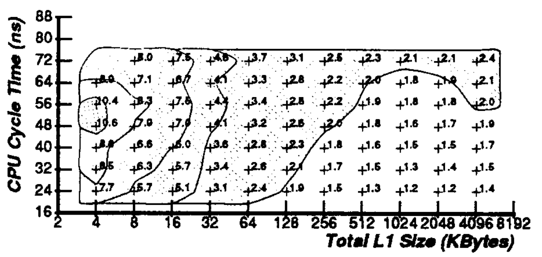
Set Size 2 Cycle Time Tradeoff

Figure 4-3



Set Size 4 Cycle Time Tradeoff

Figure 4-4



Set Size 8 Cycle Time Tradeoff

Figure 4-5

There are several important observations to be made from these graphs. First and foremost is that the numbers are almost uniformly small. Only for a total cache size of less than 16KB is the break-even point more than 6ns, which is the worst-case *data-in* to *data-out* time for an Advanced-Schottky (AS) multiplexor [17]. No data points match the 11ns *select* to *data-out* time of the same part. The conclusion is clear: it is unlikely that set associativity ever makes sense from a performance perspective for caches made of discrete TTL parts. On the other hand, in an integrated environment, where the caches are likely to be smaller, and the cost of implementing set associativity is reduced, an increase in set size may be beneficial. Secondly, the difference in break-even points between set size two and four is small: at most 2.4ns. The difference between four and eight is even smaller. This indicates that set associativity greater than two should only be implemented if it is forced by the virtual memory constraints, or if there is essentially no additional cycle time penalty.

As with the speed-size tradeoff, if a less aggressive memory system was used, the increased cache miss penalty would more significantly affect the smaller cache sizes. In that regime, the miss rate improvement brought upon by an increase in the associativity would be more significant than with the simulated memory model. The result would be larger differences between equivalent cycle times, and a greater opportunity for performance improvement through the implementation of set associativity.

We can use Figures 3-4 and 4-3 to investigate another trade-off: cache size versus set associativity. For smaller caches, we see that a doubling in size is equivalent to about 10ns of cycle time, while set associativity of 2 is worth between 4 and 8 ns. This indicates that increasing the size is clearly a more worthwhile goal that should be pursued first. However, particularly for integrated caches, increasing associativity is often an alternative when increasing the cache size is not. Also, the process of increasing the cache size may move the system to a design point at which set associativity is impractical, even though it may have originally been feasible. At the other end of the spectrum, the situation is reversed – an increase in associativity is worth more than an increase in size. However, the improvement caused by either is so small that reducing size or associativity for a decrease in cycle time would improve overall performance.

5. Main Memory Speed and Block Size

Of all the organizational parameters, the block size[10] is unique in that it directly interacts with parameters outside the cache. The choice of block size affects the cost of memory references through the miss ratio and the cache miss penalty. The focus of this section is the interaction between the block size and the memory parameters.

Increasing the block size decreases the miss ratio because of the spatial locality of references: the high probability that data in the vicinity of a referenced word is likely to be reference in the near future. Data that is further away from the current reference has a lower probability of being used 'soon'. As the fetch size[11] increases, more useful data is brought into the cache, though the average utility decreases.

In opposition to this beneficial influence on miss rate is the reduction in the number of tags with increasing block size. Fewer tags means that fewer widely separated addresses can be stored in the cache. At some point this factor invariably begins to dominate over the spatial locality and the miss rate begins to increase with further increases in block size. With respect to miss rate there is an optimum block size that is a function of the degree of locality in the program. Smith [13] has shown how the cache miss rate varies with the cache size and block size.

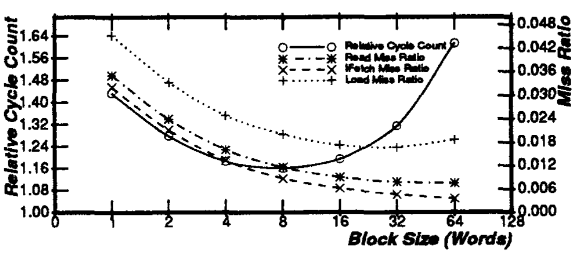
[9]As noted in Section 3, the 56ns design is abnormally inefficient, to the extent of introducing non-monotonicities in the performance as a function of cycle time. While this did not significantly affect those results, it severely distorted the analysis of set associativity. As a result, the data for the 56ns case has been smoothed to be more representative. Quantization effects can be significant and deceiving.

[10]A block is the amount of storage associated with a tag. It is sometimes called a line.

[11]The fetch size is the amount of data brought in from memory on a miss. It is also called the transfer size or sub-block [7].

The block size that optimizes system performance is significantly smaller then that which minimizes the miss rate. The memory read access time in cycles can be expressed in the form $la + BS / tr$, where la is the latency to the first word, tr is the transfer rate in words per CPU cycle and BS is the block size. This illustrates the difficulty with large block sizes, each doubling of the block size doubles the second term, to the point where it overwhelms the latency. The impact of these huge cache miss penalties can be lessened by several techniques, including early continuation (allowing the processor to continue once the desired word is received from memory), load forwarding (starting the fetch from the desired word or sub-block), or streaming (channeling incoming memory data to both the CPU and the cache). They all have the effect of increasing the performance optimal block size. Smith points out that if miss penalty is of the form $la + BS / tr$, then the block size that minimizes the mean read time[12] is only dependent on the product $la \times tr$, and not either independently. Though the read access time is the main component of memory's contribution to the total execution time, it is not the only component. It is reasonable to expect some variation from this first-order analysis.

Figure 5-1 confirms Smith's miss ratio results for the Harvard organization used in the simulations. It shows the miss ratios and relative execution time of the default organization (separate 64KB I and D caches), with a 260ns latency memory[13] The best block size on the data side is 32W, and somewhat greater than 64W on the instruction side: a reflection of the greater locality within the instruction stream. Despite this disparity, throughout this experiment both caches are consistently given the same block size.



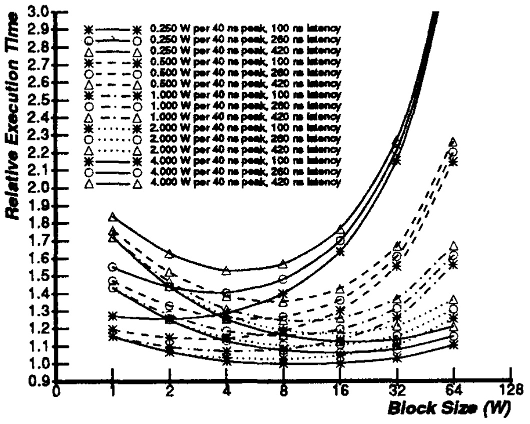
Block Size Dependencies
(1.0 W per cycle, 260 ns latency)

Figure 5-1

The other parameters in the graphs that follow are the memory latency and transfer rate. The latency is represented by the read and write operation times and the recovery time, all three of which are made equal. The latency would typically include address decoding and buffering, dynamic ram access and any error detection and correction. Here, it is varied from 100ns (three 40ns cycles) to 420ns (eleven 40ns cycles). 100ns is representative of a a very aggressive design using fast DRAMs and no ECC. The other extreme, 420ns, is in excess of a very conservative memory board to be plugged into a generic, low performance backplane. The transfer rate is varied over a range of four words in one cycle to one word in four cycles. These translate into peak bandwidths of 400MB/s to 25MB/s. For very small block sizes, having a large tr is of no benefit, as the minimum transfer time is one cycle, even if that is using only a quarter of backplane's capacity.

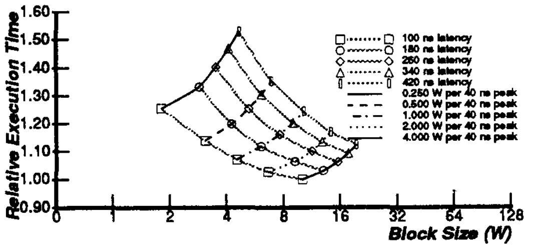
Figure 5-2 shows how the total execution time varies with the three parameters that affect the cache miss penalty. In comparison to the cache speed and size parameters, the memory system design has a relatively small impact on performance. Assuming a reasonable choice of block size, the execution time only doubles across the entire range of memory systems.

On each of the curves, representing a particular memory and backplane implementation, an optimal block size can be estimated by fitting a parabola to the lowest three points and finding its minimum. Figure 5-3 plots these minima as a function of the memory characteristics. Over most of the range, a increase in 80ns (2 cycles) in the latency causes an increase in the execution time of between 3% and 6%. Similarly, a halving of the peak transfer rate increases the execution time by between 3% and 13%. These ranges shrink slightly if the block size is restricted to binary values. The sensitivity of execution time to changes in transfer rate is largely independent of the latency, and visa versa.



Execution Time vs Memory Parameters

Figure 5-2



Optimum Block Size

Figure 5-3

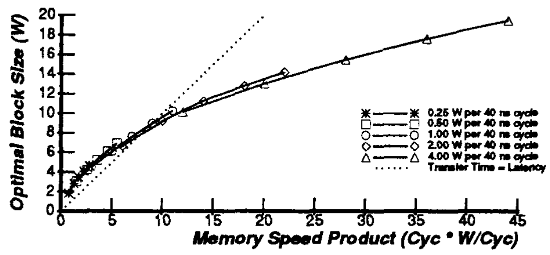
Figure 5-4 tests the first order derivation that the optimal block size is dependent only on the product of the two memory parameters. The non-integral optimal block size is plotted against the product of the latency in cycles and the transfer rate. Each curve corresponds to a single transfer rate, with the points representing the latencies of 3, 5, 7, 9 and 11 cycles. The line segments line up quite well, verifying that the optimal block size is a function of the memory speed product, $la \times tr$. Since the block size of choice is solely a function of this product, as DRAM and backplane technologies improve, their influences tend to cancel, leaving the best blocksize unchanged.

An experienced engineer might expect that a good design point would be to equalize the transfer time and the latency. The dotted line on Figure 5-4 shows the block size choice that balances the two components of the miss penalty. It's clear that the optimum block size does not follow this strategy. When the memory product is small and the memory technology is superior to the backplane's, the optimum block size is larger than

[12]The average memory read time is $(la + BS / tr) \times MR(BS)$ where $MR(BS)$ is the miss rate as a function of the block size. Equating its derivative with respect to block size to zero determines the optimal block size as a function of the memory parameters.

[13]A 260ns latency makes for a 12 cycle read request for a block size of 4 and a cycle time of 40ns.

expected. In the region above the dotted line, more than half the time is spent transferring data. Similarly, when the product is high, indicating that the memory technology is poor with respect to the backplane, the optimal block size is smaller than one might expect. Below the dotted line, more than half the time is spent waiting for memory to present the first word.



Optimum Block Size versus Speed Ratio

Figure 5-4

The primary conclusion to be drawn from this section is that without special mechanism to reduce the cache miss penalty, the optimal block sizes are quite small and relatively invariant. For the central portion of the design space, the best binary block size was either four or eight words. Since the transfer rate and latency are inversely linked to the cycle time, and the optimal block size only a function of *la* x *tr*, the block size of choice is completely independent of the CPU cycle time.

6. Multilevel Caches

Embedded in these results are two implications for computer design. First, once the cache size has been increased to 64KB level, there is little room for improvement. An implementor apparently has no way to significantly increase the performance of the memory hierarchy and system through the application of more hardware. Increasing the cache size or adding set associativity at the expense of cycle time is likely to be detrimental. Exchanging the cache RAMs for more copies of a smaller yet faster variety may not reduce the cycle time if additional multiplexors or additional buffering is needed as a result of the switch.

The second implication has to do with the evolution of computer organization in the face of continued improvements in implementation technologies. A fundamental question confronting computer implementors is what sort of memory hierarchy is needed to keep high performance machines running at close to their peak rates. The mean cycles per reference must be kept reasonably low in the face of decreasing cycle times. Figure 3-4 indicates that for high speed CPUs, it is advantageous to increase the cycle time to obtain a bigger cache. Small, high speed caches tuned to trim, efficient RISC implementations do not appear to be a good design point. This conclusion directly confronts the designer's desire to increase performance by squeezing the cycle time.

If the entire system scales evenly, the basic tradeoffs do not change. If all the temporal parameters are divided by a common factor, the shape and position of the curves remain the same while the slopes, expressed in nanoseconds per doubling, scale down. Expressed as a fraction of the cycle time per doubling, the slopes remain constant. Unless there is a dramatic shift in the proportion of the cycle time needed to move from one RAM size to the next, the optimal cache size is not likely change much as technologies improve, contingent on the main memory access time scaling as well.

Unfortunately, it is difficult to scale main memory access times with the CPU cycle time. Chip to chip communication and control logic already make up a significant fraction of the latency period. Buffering and decoding will take up a larger fraction of the latency as dynamic RAM access times decrease. Transfer rates are limited by physical dimensions and fundamental electrical properties of materials. Adding width to data paths quickly becomes impractical due to the number of wires involved. Since both the main memory latency and transfer time will scale less than linearly with improvements in integrated circuit technologies, the cache miss penalties, expressed in cycles, will increase as cycle times continue to decrease. This in turn favours larger caches and set associativities. The overall effect is to further drive the designer away from the goal of a tight CPU cycle time and a small cache to match.

Fortunately, there is another way to interpret the data – one which exposes the solution. The hidden variable in the plots of the speed-size design space is cache miss penalty. As the cycle time was varied from 20ns though 80ns, the cache miss penalty went from 14 to 8 cycles. Table 3 rephrases the speed – cycle time trade off in terms of cache miss penalty. For each cache size, the first column is the total total cycle count divided by the number of references. Since there are two caches, the value drops below one for large caches. The second number is the cycle time degradation equivalent to a doubling of a cache size expressed as a fraction of the CPU cycle time. For instance, in Figure 3-4 the slope for the constant performance curve for a 24ns, 4KB cache was 11.5ns. So when the cache miss penalty was 13 cycles, a doubling of the cache size was worth 48% of the cycle time.

	Cache Size							
	4 KB		16 KB		64 KB		256 KB	
	Cycles per		Cycles per		Cycles per		Cycles per	
Cycles per Read	Ref.	sizex2	Ref.	sizex2	Ref.	sizex2	Ref.	sizex2
13	2.49	0.48	1.64	0.28	1.17	0.16	0.95	0.05
12	2.37	0.44	1.58	0.27	1.14	0.15	0.94	0.04
11	2.19	0.37	1.49	0.23	1.10	0.14	0.92	0.04
10	2.08	0.32	1.43	0.19	1.08	0.12	0.91	0.04
9	1.92	0.34	1.35	0.20	1.04	0.11	0.89	0.03
8	1.75		1.27	0.15	1.00	0.10	0.88	0.03

Table 3: Memory Performance versus Cache Miss Penalty

The table makes two points. The first is that for small caches, with their high miss ratios, the cycles per reference is a strong function of the miss penalty. This says that to keep the cycles per reference low, the trend of increasing cache miss penalties must be reversed. The second and more subtle point is that the fraction of a cycle that is equivalent to a doubling of cache size increases with the cycles per memory read. This means that as the miss penalty decreases, it is less likely that a doubling of the cache size will be beneficial.

The conclusions to be drawn from this perspective of a fixed cycle time and varying miss penalty are two fold. First, reducing the miss penalty decreases the optimum cache size. If the cache miss penalty is short enough, the optimum cache size and cycle time will be reduced to the point where the CPU is again limiting the system cycle time. Second, since the cycles per reference is approximately a linear function of the miss penalty, any desired memory hierarchy performance goal can be attained by sufficiently reducing the miss penalty, regardless of the cache size. These conclusions are independent of the actual CPU cycle time, and apply equally for 80ns systems and 5ns systems.

The fundamental question – how to get some desired performance level out very short cycle time machine – becomes "what cache miss penalty is required?" Not surprizingly, the solution to the problem of building a memory system that responds in three or five 10ns cycles is identical to the answer the historical need to reduce main memory latency: a cache. Designing a second cache between the CPU/cache and main memory poses the same set of questions as the first level of caching, but with a different set of parameters, constraints and goals. The overall conclusion is that as the disparity between main memory times and CPU cycle time continues to grow, the only way to deliver a consistent proportion of the peak CPU performance is through the use of a multilevel cache hierarchy.

7. Conclusions

Though uniprocessor cache design has been extensively studied in the past, the focus has primarily been miss ratios and transfer rates. These metrics only give a partial picture of a cache's performance, since temporal issues are not considered. Introducing timing parameters leads to a new set of tradeoffs in cache design. This paper has examined three of these: cache size versus CPU cycle time, set associativity versus cycle time, and block size versus memory speed.

The most striking result is that the performance of a fast processor with a small fast cache can be improved by making a relatively large increase in cycle time, if this allows the caches to be made bigger. For the Harvard organization and aggressive memory design simulated, the benefit of doubling the cache ranged from over 10ns for small (16KB) caches to less than 2.5ns for large (256KB) caches. The size of the tradeoff in nanoseconds per doubling in cache size is essentially independent of the cycle time of the system. This means a 50ns 64KB machine performs better than a 40ns 16KB machine. Making the cache extremely large is also not beneficial. As the cache size increases, the equivalent degradation in cycle time lessens. If the cache is too big, significant performance improvement can be had by reducing its size and shrinking the cycle time. The combined result is that across a wide range of systems, the optimal direct mapped total cache size is between 32KB and 128KB.

Unless mandated by virtual memory constraints, it is unlikely that any set associativity is warranted in caches made of discrete TTL and static RAMs. The cycle time degradation over a direct mapped implementation caused by the comparison and selection hardware would overwhelm the benefit derived from the improved miss rate. For integrated caches below 16KB, the benefit may be sufficiently large that the cycle time penalty caused by set associativity is less than the break-even point. For smaller caches, increases in size have a much more significant impact on performance than the addition of set associativity.

The cache miss penalty is dependent on the block size, the memory latency and the transfer rate. For a wide variety of memory and bus speeds, the optimal block size is 4 or 8 words. The optimal block size is strictly a function of the product of the memory latency and the transfer rate, and is not dependent on the CPU cycle time. The overall performance is a much weaker function of block size than it is of cache size or cycle time.

These tradeoffs in cache design strongly indicate that ordinary caches have a performance limit. With ever improving integrated circuit technologies, the most effective, and perhaps only way to keep the memory component of the overall execution time from growing disproportionately will be to use multi-level cache hierarchies. The existence of a second level cache modifies the speed – size tradeoff for the first level cache by reducing the cost of first-level cache misses, making small, fast caches a viable alternative.

References

1. Agarwal, A., Sites, R., Horowitz, M. ATUM: A New Technique for Capturing Address Traces Using Microcode. Proc. 13th Sym. on Computer Architecture, IEEE/ACM, Tokyo, Japan, June, 1986.

2. Agarwal, A. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Ph.D. Th., Stanford University, Computer Systems Laboratory, May 1987. Available as CSL-TR-87-332.

3. Agarwal, A., Hennessy, J., and Horowitz, M. Cache Performance of Operating System and Multiprogramming Workloads. Submitted for publication.

4. Duncombe, R.R. The SPUR Instruction Unit: An On-Chip Instruction Cache Memory for a High Performance VLSI Multiprocessor. Computer Science Division 87-307, University of California, Berkeley, August, 1986.

5. Goodman, J.R. Coherency For Multiprocessor Virtual Address Caches. ASPLOS-II, IEEE, October, 1987, pp. 72-81.

6. Haikala, I.J., and Kutvonen, P.H. Split Cache Organizations. Performance '84, 1984, pp. 459-472.

7. Hill, M.D., and Smith, A.J. Experimental Evaluation of On-Chip Microprocessor Cache Memories. Proceedings of the 11th Annual Symposium on Computer Architecture, June, 1984, pp. 158-166.

8. Hill, M.D. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. Th., University of California, Berkeley, November 1987.

9. Horowitz, M., Chow, P., Stark, D., Simoni, R., Salz, A., Przybylski, S., Hennessy, J., Gulak, G., Agarwal, A. and Acken, J. "MIPS-X: A 20 MIPS Peak, 32-Bit Microprocessor with On-Chip Cache". *Journal Solid State Circuits SC-22*, 5 (October 1987).

10. Micheal M.J., et. al. "Helwett-Packard Precision Architecture: The Processor". *Hewlett-Packard Journal 37*, 8 (August 1986), 4-22.

11. Smith, A.J. "Cache Memories". *ACM Computing Surveys 14*, 3 (September 1982), 473-530.

12. Smith, A.J. Cache Evaluation and the Impact of Workload Choice. Proceedings of the 12th Annual Symposium on Computer Architecture, June, 1985, pp. 64-73.

13. Smith, A.J. "Line (Block) Size Choice for CPU Cache Memories". *IEEE Transaction on Computers C-36*, 9 (September 1987), 1063-1075.

14. Smith, A.J. Design of CPU Cache Memories. Computer Science Division 87-357, University of California, Berkeley, June, 1987.

15. Smith, A.J. "A Comparative Study of Set Associative Memory Mapping Algorithms And Their Use for Cache and Main Memory". *IEEE Transactions on Software Engineering SE-4*, 2 (March 1978), 121-130.

16. Smith, A.J. "Bibliography and Readings on CPU Cache Memories and Related Topics". *Computer Architecture News 14*, 1 (January 1986), 22-42.

17. *ALS/AS Logic Data Book*. Texas Instruments, Dallas, Texas, 1986.

18. Tucker, S.G. "The IBM 3090 System: An Overview". *IBM Systems Journal 25*, 1 (1986), 4-19.