The Effects of Size, Associativity, and Replacement Policy on the MIPS L1 Data Cache

Javier Lombillo
2016-04-13
CET3126C Lab4
School of Engineering & Technology
Miami-Dade College

Abstract

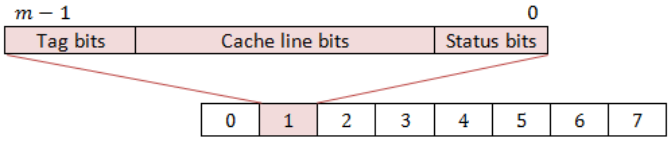
Computer storage may be optimized for size, speed, cost, or some combination of the three, though no single combination satisfies every storage requirement. As such, computer architectures are designed with separate storage components, organized in a hierarchy in which the size of the storage is proportional to the distance from the processor, and the speed of the storage is inversely proportional. At the lowest level of the hierarchy, closest to the processor, registers represent the fastest and smallest storage units. Next come the caches, themselves organized in a miniature hierarchy, followed by main memory and secondary storage.

In this paper, cache memory—specifically, the lowest-level L1 data cache—is analyzed against three benchmarks, varying the parameters of cache size, associativity, and replacement policy.

1. Introduction

Functionally, a memory cache can be viewed as a one-dimensional array, with each row (or column) representing a *cache slot*, indexed sequentially by position. Each slot contains a *cache line* for data storage, typically 32 bytes in size, as well as status and tag bits. The status bits indicate the state of the cache line, whether it is valid (all cache lines contain garbage on first access), or if it is "dirty" and needs to be written to secondary storage before it can be replaced. The tag bits are used to identify the memory address to which the cache line belongs. The general structure of an m-bit cache slot is illustrated below in Figure 1:

Figure 1: Cache slot



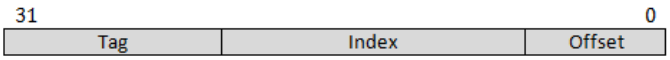
In the MIPS architecture, memory is byte-addressable and word-aligned on four-byte boundaries. The size of the cache line can vary between implementations, but for concreteness this paper assumes 32 bytes. Thus, a cache line can store four words of byte-addressable data. Because the purpose of cache is to make slower main memory appear to the processor as if it is very fast internal memory, the data in a cache line are associated with an address from main memory. This is the address of the first byte in the 32-byte block of main memory. As such, when this paper refers to a *block*, it should be understood to mean a 32-byte contiguous chunk of main memory. Likewise, when this paper refers to an *address*, it should be understood that the first byte of a block is being referenced. *Cache line* always refers to the 32-byte contiguous chunk of cache memory used to store data in a cache slot.

To belabor the point, one may say that addresses are to cache slots as blocks are to cache lines.

Direct-mapped Caches

The simplest method of associating a block with a cache line is directly, that is, each address is mapped to precisely one cache slot. Such a scheme is called a **direct-mapped cache**. To accomplish the mapping, a subset of bits from the address is used to index into its corresponding cache slot; therefore, an n-slot cache requires $\log_2 n$ index bits. The image below illustrates how a word-length address is parsed:

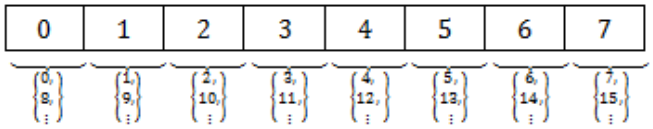
Figure 2: Block address



The offset bits in the address provide byte-level access to the cache line; thus, for m-byte cache lines, $\log_2 m$ bits are used for the offset. Because the size of the block address space is significantly larger than the number of slots in the cache, the direct-mapped scheme results in a many-to-one mapping. To differentiate between the various addresses mapped to a slot, the address's tag bits are stored in the slot along with the block data, as shown in Figure 1. In this way, one slot can be responsible for multiple blocks of data, even though at any given time it can store only one block.

Figure 3 illustrates the address-to-slot mapping of an 8-slot cache:

Figure 3: Direct-mapped cache



The values below each slot represent memory addresses whose index bits map to that particular slot. When the processor wants to accesses data memory—e.g., with a load instruction—it does so through the L1 cache. The appropriate

slot is indexed by the index bits in the address, and a *cache hit* occurs if the tag bits match and the valid bit is set. Otherwise a *cache miss* results: the desired data is fetched from main memory, replacing the previously stored data, which has essentially been *evicted* from the cache.

Note that because of the strict mapping, index collisions—wherein multiple addresses are contending for the same slot—always result in a cache line eviction, regardless of the availability of other slots. Such a scheme is biased to have high collision rates unless the requested addresses are uniformly distributed across the address space, which is rarely the case.

One way to mitigate this issue is to decouple addresses from cache slots, instead associating each address with a *set* of cache slots. Because address collisions occur at the set level, and each set contains multiple slots, a collision does not necessarily result in a cache line eviction—another slot in the set can be used if it is available. Furthermore, even when all slots in a set are in use, a cache replacement policy can better manage which line should be evicted, such as the least recently used. Caches that employ such an address-to-set mapping are called *associative*.

Associative Caches

Associativity describes the distribution of block addresses over sets of cache slots. In a k-associative cache with S total slots, there are S/k total sets, with k slots per set. For example, a 2-associative cache with 256 slots has 128 sets, each containing 2 slots. Associativity increases as k approaches S. At the upper bound, where $k = S$, the cache is **fully associative**: one set contains the entire cache. Because addresses are coupled to sets, and there is only one set, a block can be stored *anywhere* in the cache.

The other extreme occurs when $k = 1$, resulting in a direct-mapped cache, as described above. In this case there are $S/1 = S$ total sets, each comprised of 1 slot.

Intuitively, the larger the set—i.e., the more cache slots in each set—the greater the freedom to place a block. The address-to-set maps are fixed, but the block-to-line maps are not. For example, consider a 4-associative cache with 8 slots; such a cache is partitioned into $8/4 = 2$ sets. The cache can be visualized as a one-dimensional 8-element array, with one set comprised of the first four elements, and the other set the last four. Define the map that sends an address n to one of the sets as

$$f(n) = n \bmod 2.$$

Then, for some N number of blocks, those with even addresses will map to the first set, and those with odd addresses to the second set:

Figure 4: 4-associative cache

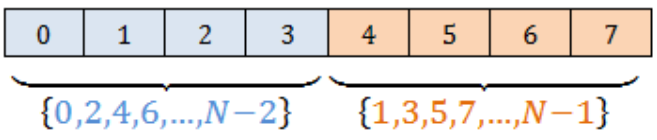
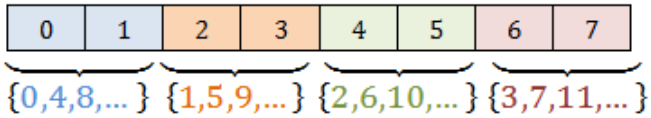


Figure 5 presents a visualization of the same cache with a 2-associative mapping, using the map $f(n) = n \bmod 4$:

Figure 5: 2-associative cache



In both cases, the probability of collision is equal, fixed by the ratio S/N, and so is independent of associativity. However, the *utilization* of the cache—i.e., the percentage of slots in use before a cache line must be evicted—depends on the number of slot *options* available. In a direct-mapped cache ($k = 1$), there are no options: each address is mapped to one and only one cache slot; any collision results in an eviction, regardless of the availability of other slots. This results in poor utilization. On the other end, in a fully associative cache ($k = S$), *every* slot is an option, resulting in maximum utilization.

More generally, in a k-associative, S-slot cache serving N blocks, each set contains

$$M = \frac{N}{S/k} = \frac{kN}{S}$$

possible mappings, each of which can be associated with any of the k slots in the set. To make this concrete, consider a 2-slot set that is associated with the four addresses $\{0, 1, 2, 3\}$. There are 12 possible ways for the two slots to be filled:

$$(0,1), (0,2), (0,3),$$
$$(1,0), (1,2), (1,3),$$
$$(2,0), (2,1), (2,3),$$
$$(3,0), (3,1), (3,2)$$

where the ordered pairs (m, n) represent the addresses in each slot. Mathematically, this corresponds to a 2-permutation of a 4-set, resulting in

$$\frac{4!}{(4-2)!} = 12$$

degrees of freedom in filling the slots. Generally, for any M-set, there are

$$d = \frac{M!}{(M-k)!}$$

possible permutations of the set, which can be thought of as d degrees of slot-mapping freedom.

For example, let $N = 64$ total blocks. Then, in the 2-associative case, there are

$$M = \frac{2 \times 64}{8} = 16$$

mappings per set. This results in

$$\frac{16!}{(16-2)!} = 240$$

degrees of freedom. In contrast, the 4-associative cache has

$$M = \frac{4 \times 64}{8} = 32$$

mappings per set, giving it

$$\frac{32!}{(32-4)!} = 863{,}040$$

degrees of freedom, a substantial improvement. Note that these numbers are not meaningful in themselves, rather they are meant to convey a sense of the relative power of increasing associativity levels for some fixed cache size.

2. Background

The workloads of typical desktop and server applications demand gigabytes of fast random-access memory. Ideally, this memory is as fast as the processor, requiring at most one or two clock cycles to complete. However DRAM technology—which has the density and power characteristics that make multi-gigabyte RAM practical—is too slow to be clocked by the sub-nanosecond cycle periods used in modern processors. As such, DRAM access requires hundreds of processor cycles to complete, presenting a significant performance bottleneck.

SRAM technology can be made nearly as fast as the processor, with access requiring only a few clock cycles [1]. The critical limitation with SRAM is memory density. In DRAM, a bit of memory is stored by a tiny capacitor (on the order of *femto*farads [2]), guarded by a single transistor for read and write operations. This simplicity allows for highly dense arrays of bits, making DRAM inexpensive and relatively power efficient. The cost of this hardware simplicity is interface complexity; reads from DRAM are destructive—the capacitor storing the bit is slightly discharged on each read—and capacitors leak charge over time. As such, DRAM requires periodic refreshing of its state, a maintenance routine that affects performance and complicates its interface. A further limiting factor in DRAM speed is the non-zero charge/discharge time of capacitors.

In contrast, the bits in an SRAM array are stored solely with transistors. Typically, an SRAM cell is comprised of six CMOS transistors, four of which are arranged as two cross-coupled inverters to store the bit, and two serving as access transistors for reads and writes [3]. This bi-stable topology makes the bits more electrically robust than capacitive storage. Consequently, the hardware interface is significantly simpler than DRAM, and therefore faster. The tradeoff is reduced bit density and increased power consumption, both of which result in an increased cost per bit.

These factors make DRAM appropriate for main memory and SRAM for cache memory.

3. Experimental Methodology

Several simulations were performed on three SPEC benchmarks, varying the L1 data cache parameters of size, associativity, and replacement policy. At the end of each run, the simulation software reported several performance statistics, including the L1 data cache miss rate. These miss rates were tabulated, converted into hit rates, and a comparative analysis was performed.

3.1. Simulation Framework

The simulation software used was sim-cache, a functional cache simulator, from the SimpleScalar v3.0 suite. The platform modeled was the 32-bit Alpha ISA.

3.2. Simulated Workloads

Cache performance was evaluated using three SPEC benchmarks, **gcc**, **Go**, and **Anagram**.

Unless otherwise noted, the L1 data cache parameters were set as following:

Cache size	256 slots
Cache line size	32 bytes
Associativity	Direct-mapped
Replacement policy	LRU

All other sim-cache options were left at their default values.

NB: The author observed small discrepancies in the Anagram simulation statistics depending on whether the output of sim-cache was piped to another process or displayed directly to the terminal (stdout). The discrepancies—on the order of thousandths of a percent—were not deemed significant. The SimpleScalar documentation acknowledges the issue: "redirecting output will cause subtle changes in printf() execution" [3].

4. Results

4.1. Replacement Policy

Three replacement policies were tested:

1. Least-recently used (LRU)
2. First in, first out (FIFO)
3. Random eviction

The names of the policies are sufficiently self-descriptive to not require further explanation. The choice of replacement policy had no effect in a direct-mapped cache. This is an intuitive result, given the zero degrees of mapping freedom inherent in the scheme. To compare the effectiveness of the policies, the three benchmarks were twice simulated with each of the policies, once with a fully associative cache and once with a 2-associative cache. In every run, cache size was 1 kilobyte (32 cache slots). The resulting hit rates are summarized in Table 1 below:

Table 1: Replacement policies

	gcc	
2-assoc.	LRU	87.77%
	FIFO	86.92%
	Random	86.44%
Full	LRU	98.35%
	FIFO	98.11%
	random	98.04%

	Anagram	
2-assoc.	LRU	94.70%
	FIFO	94.32%
	random	94.53%
Full	LRU	99.62%
	FIFO	99.60%
	random	99.59%

	Go	
2-assoc.	LRU	81.46%
	FIFO	80.15%
	random	79.66%
Full	LRU	99.88%
	FIFO	99.83%
	random	99.82%

In all cases, the LRU policy performed best. Perhaps surprisingly, the largest hit rate differentials occurred in the 2-associative runs; one might expect that differences in policies would better manifest with the greater freedom of a fully associative cache. The author believes, however, that the maximal utilization inherent with full associativity left less room, so to speak, for the policies to differentiate themselves. Overall, the effect of replacement policy on cache hits was small compared to the other parameters.

4.2. Cache Size

Five cache sizes were compared using the direct-mapped scheme. As expected, the smallest 32-slot (1 KB) cache performed worst. Hit rate consistently improved for each benchmark with increasing cache size, though the incremental improvement rate quickly decreased after 2048 slots (64 KB), as shown in Figure 6 below:

Figure 6: Cache size comparison

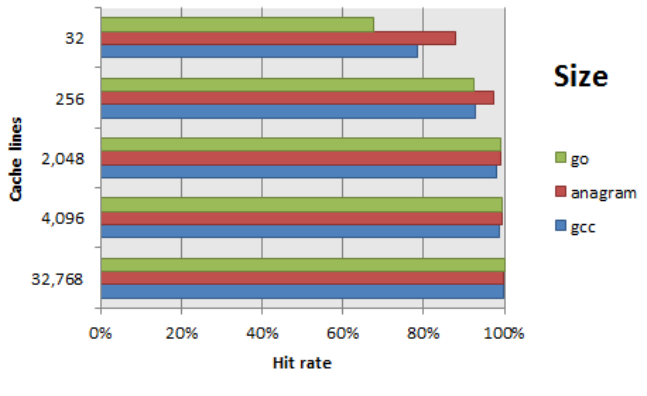
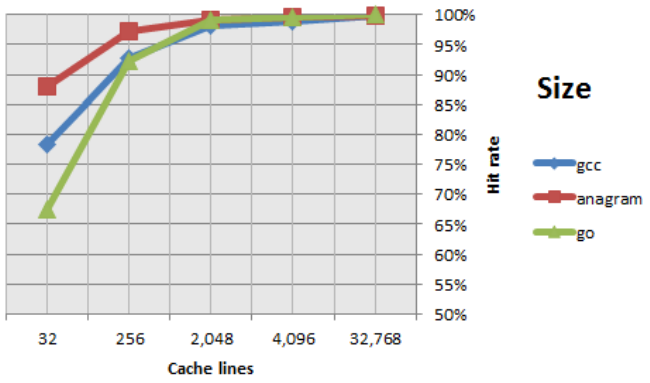


Figure 7 better illustrates the logarithmic approach of hit rates to the 100% asymptote:

Figure 7: Cache size comparison, log-linear scale



Incremental improvement by percentage for each cache size was calculated in Table 2 below. The diminishing returns of cache size in a direct-mapped scheme are plainly evident.

Table 2: Incremental improvement

Cache lines	gcc	Anagram	Go
32768 (8x)	0.97%	0.28%	0.36%
4096 (2x)	0.77%	0.38%	0.42%
2048 (8x)	5.37%	1.91%	6.86%
256 (8x)	14.44%	9.28%	24.83%

4.3. Associativity

To test the effect of associativity on hit rate relative to cache size, the benchmarks were simulated on three increasingly larger caches, switching between direct-mapped and fully associative schemes. The power of full association was evident: for all three benchmarks, the increase from 8 KB to 32 KB provided *no* improvement with the fully associative cache, yet showed significant improvement with the direct-mapped cache. This strongly suggests that hit rate maxima are achieved at smaller cache sizes with full associativity than with the direct-mapped scheme.

The results for each benchmark are given below:

Figure 8: Full vs direct, gcc

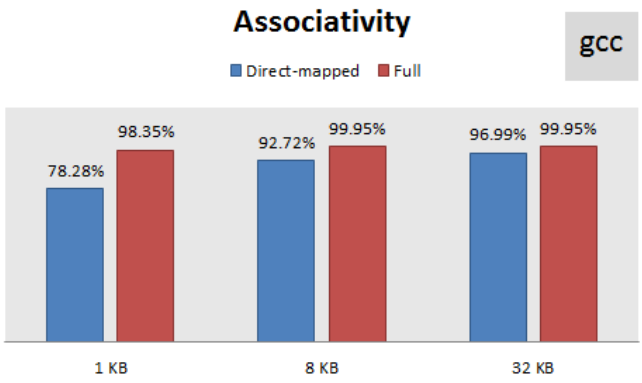
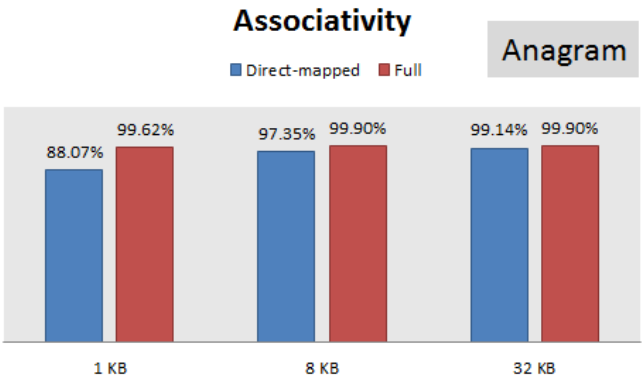
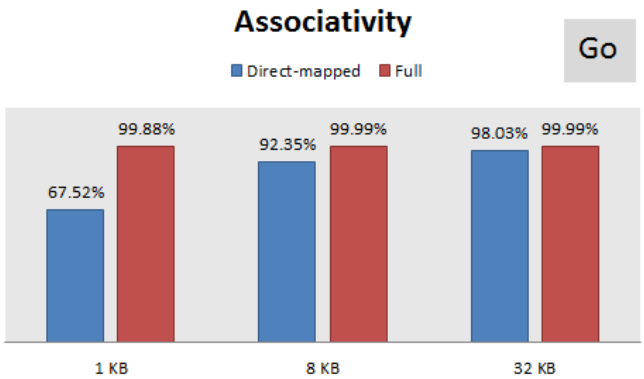


Figure 9: Full vs direct, Anagram



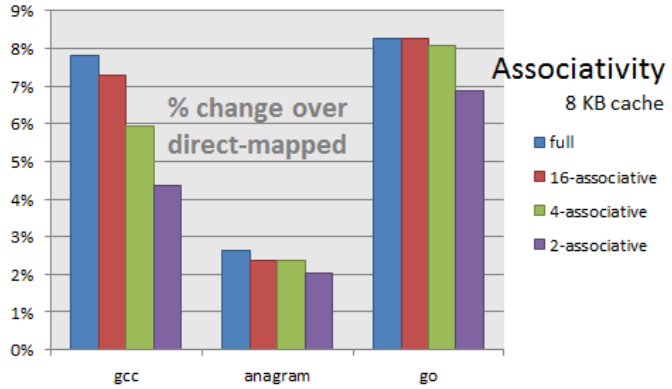
The largest total improvement was observed on the 1 KB cache with the Go benchmark, shown below. Changing to a fully associative scheme resulted in a better than 32% hit rate improvement. More impressive still, the difference between the fully associative 1 KB cache and the fully associative 32 KB—a 32x increase in size—was only 0.11%, further demonstrating the impact of full associativity.

Figure 10: Full vs direct, Go



To compare the relative performance of the intervening associativity levels, the benchmarks were run again on an 8 KB cache, each time varying associativity, with performance expressed as the percent difference over the baseline (direct-mapped) hit rate for each benchmark. The results are shown in Figure 11 below:

Figure 11: Comparison of associativity levels, 256-slot cache

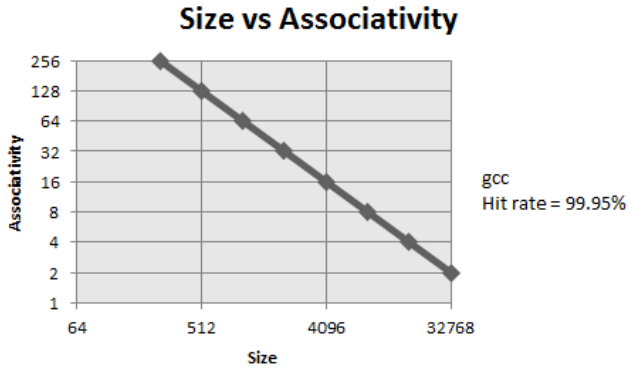


The gcc benchmark was most sensitive to associativity level, though in general the pattern of diminishing returns remained. It is interesting to note that in all cases, the 16-associative cache performed nearly as well as the fully 256-associative cache, suggesting that 16-associativity may represent the sweet spot in terms of performance and hardware complexity.

4.4 Size vs Associativity

To explore the relationship between associativity and cache size, several simulations were performed, varying each parameter and noting the resulting hit rate. A log-log relationship was noted: for a given hit rate, a stepwise decrease in associativity required an equivalent stepwise increase in cache size; that is, lowering the associativity level by a factor of 2^{-n} required raising the cache size by a factor of 2^n to achieve the same hit rate. For example, with the gcc benchmark, a (fully) 256-associative, 256-slot cache scored a 99.95% hit rate. Likewise, the same hit rate was scored for a 128-associative, 512-slot cache; a 64-associative, 1024-slot cache; and so on.

Figure 12: Size vs associativity, gcc, fixed hit rate



It should be noted that this linearity (on log scales) breaks down at $k = 1$, the direct-mapped scheme. No matter how large the cache was made (the author stopped trying at over 8 million slots, a highly impractical 268 MB L1 cache), the hit rate would not climb over 99.94%. In any case, the tight relationship between size and associativity for a given hit rate indicates that, in theory, one can be traded for the other in

pursuit of a target hit rate. Unfortunately, both size and associativity are strongly constrained by hardware considerations.

5. Conclusion

The results presented in this paper describe cache hit rates as a percentage, which has the unfortunate effect of making the numbers appear less significant than they actually are. Though a 5% improvement may seem small, its magnitude is made clear in the context of a processor executing millions of instructions per second. To put this into perspective, assume that 25% of the instructions in an arbitrary workload require memory access. Further assume that a cache miss results in a 100-cycle delay (a conservative value). This implies that in a set of a million instructions, 250,000 will try to access memory. If the cache hit rate is 90%, then 25,000 instructions will incur a 100-cycle penalty, requiring an extra 2.5 million cycles to complete. Increasing the hit rate "only" 5% reduces the number of extra cycles by *half*, to 1.25 million. Clearly every percent counts.

So why not make the L1 cache 100 MB in size and fully associative? With respect to size, the L1 cache is located inside the real-estate constrained processor chip. The low bit-density of fast SRAM technology prevents cache size from scaling like DRAM main memory per unit area. In terms of associativity, the hardware required to perform a parallel tag search on every slot quickly becomes impractical as the number of cache slots per set increases. Even a modest 256-slot cache would need 256 comparators, presenting an enormous cost to the designer's real-estate and power budget. As such, only small, specialized caches—such as translation-lookaside buffers—are fully associative.

Fortunately, the simulations demonstrate that *any* level of associativity results in a significant cache hit improvement over the simplistic direct-mapped scheme. Furthermore, the difference in hit performance between an impractical fully associative and a practical 16-associative cache is minimal. Therefore, the optimal L1 data cache is as large as the design budget will allow, 16-associative, using an LRU replacement policy.

6. References

[1] http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html
[2] http://www.cs.berkeley.edu/~pattrsn/294/LEC9/lec.html
[3] Pavlov, Andrei, and Manoj Sachdev. CMOS SRAM circuit design and parametric test in nano-scaled technologies: process-aware SRAM design and test, p. 15. Vol. 40. Springer Science & Business Media, 2008.
[4] http://www.simplescalar.com/faq.html, Q2.