$$2^{\text{index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

$$\text{CPU execution time} = (\text{CPU clock cycles} + \text{Memory stall cycles}) \times \text{Clock cycle time}$$

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

$$\text{Memory stall cycles} = \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

$$\frac{\text{Misses}}{\text{Instruction}} = \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}}$$

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Memory stall clock cycles}}{\text{Instruction}} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\text{CPU execution time} = \text{IC} \times \left(\text{CPI}_{\text{execution}} + \text{Miss rate} \times \frac{\text{Memory accesses}}{\text{Instruction}} \times \text{Miss penalty} \right) \times \text{Clock cycle time}$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}}{\text{Instruction}} \times (\text{Total miss latency} - \text{Overlapped miss latency})$$

$$\text{Average memory access time} = \text{Hit time}_{\text{L1}} + \text{Miss rate}_{\text{L1}} \times (\text{Hit time}_{\text{L2}} + \text{Miss rate}_{\text{L2}} \times \text{Miss penalty}_{\text{L2}})$$

$$\frac{\text{Memory stall cycles}}{\text{Instruction}} = \frac{\text{Misses}_{\text{L1}}}{\text{Instruction}} \times \text{Hit time}_{\text{L2}} + \frac{\text{Misses}_{\text{L2}}}{\text{Instruction}} \times \text{Miss penalty}_{\text{L2}}$$

Figure B.7 Summary of performance equations in this appendix. The first equation calculates the cache index size, and the rest help evaluate performance. The final two equations deal with multilevel caches, which are explained early in the next section. They are included here to help make the figure a useful reference.

B.3 Six Basic Cache Optimizations

The average memory access time formula gave us a framework to present cache optimizations for improving cache performance:

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Hence, we organize six cache optimizations into three categories:

- *Reducing the miss rate*—larger block size, larger cache size, and higher associativity
- *Reducing the miss penalty*—multilevel caches and giving reads priority over writes
- *Reducing the time to hit in the cache*—avoiding address translation when indexing the cache

Figure B.18 on page B-40 concludes this section with a summary of the implementation complexity and the performance benefits of these six techniques.

The classical approach to improving cache behavior is to reduce miss rates, and we present three techniques to do so. To gain better insights into the causes of misses, we first start with a model that sorts all misses into three simple categories:

- *Compulsory*—The very first access to a block *cannot* be in the cache, so the block must be brought into the cache. These are also called *cold-start misses* or *first-reference misses.*
- *Capacity*—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
- *Conflict*—If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called *collision misses*. The idea is that hits in a fully associative cache that become misses in an *n*-way set-associative cache are due to more than *n* requests on some popular sets.

(Chapter 5 adds a fourth C, for *coherency* misses due to cache flushes to keep multiple caches coherent in a multiprocessor; we won't consider those here.)

Figure B.8 shows the relative frequency of cache misses, broken down by the three C's. Compulsory misses are those that occur in an infinite cache. Capacity misses are those that occur in a fully associative cache. Conflict misses are those that occur going from fully associative to eight-way associative, four-way associative, and so on. Figure B.9 presents the same data graphically. The top graph shows absolute miss rates; the bottom graph plots the percentage of all the misses by type of miss as a function of cache size.

To show the benefit of associativity, conflict misses are divided into misses caused by each decrease in associativity. Here are the four divisions of conflict misses and how they are calculated:

- *Eight-way*—Conflict misses due to going from fully associative (no conflicts) to eight-way associative
- *Four-way*—Conflict misses due to going from eight-way associative to four-way associative
- *Two-way*—Conflict misses due to going from four-way associative to two-way associative
- *One-way*—Conflict misses due to going from two-way associative to one-way associative (direct mapped)

As we can see from the figures, the compulsory miss rate of the SPEC2000 programs is very small, as it is for many long-running programs.

Having identified the three C's, what can a computer designer do about them? Conceptually, conflicts are the easiest: Fully associative placement avoids all

Cache size (KiB)	Degree associative	Total miss rate	Miss rate components (relative percent) (sum = 100% of total miss rate)					
			Compulsory		Capacity		Conflict	
4	1-way	0.098	0.0001	0.1%	0.070	72%	0.027	28%
4	2-way	0.076	0.0001	0.1%	0.070	93%	0.005	7%
4	4-way	0.071	0.0001	0.1%	0.070	99%	0.001	1%
4	8-way	0.071	0.0001	0.1%	0.070	100%	0.000	0%
8	1-way	0.068	0.0001	0.1%	0.044	65%	0.024	35%
8	2-way	0.049	0.0001	0.1%	0.044	90%	0.005	10%
8	4-way	0.044	0.0001	0.1%	0.044	99%	0.000	1%
8	8-way	0.044	0.0001	0.1%	0.044	100%	0.000	0%
16	1-way	0.049	0.0001	0.1%	0.040	82%	0.009	17%
16	2-way	0.041	0.0001	0.2%	0.040	98%	0.001	2%
16	4-way	0.041	0.0001	0.2%	0.040	99%	0.000	0%
16	8-way	0.041	0.0001	0.2%	0.040	100%	0.000	0%
32	1-way	0.042	0.0001	0.2%	0.037	89%	0.005	11%
32	2-way	0.038	0.0001	0.2%	0.037	99%	0.000	0%
32	4-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
32	8-way	0.037	0.0001	0.2%	0.037	100%	0.000	0%
64	1-way	0.037	0.0001	0.2%	0.028	77%	0.008	23%
64	2-way	0.031	0.0001	0.2%	0.028	91%	0.003	9%
64	4-way	0.030	0.0001	0.2%	0.028	95%	0.001	4%
64	8-way	0.029	0.0001	0.2%	0.028	97%	0.001	2%
128	1-way	0.021	0.0001	0.3%	0.019	91%	0.002	8%
128	2-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	4-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
128	8-way	0.019	0.0001	0.3%	0.019	100%	0.000	0%
256	1-way	0.013	0.0001	0.5%	0.012	94%	0.001	6%
256	2-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	4-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
256	8-way	0.012	0.0001	0.5%	0.012	99%	0.000	0%
512	1-way	0.008	0.0001	0.8%	0.005	66%	0.003	33%
512	2-way	0.007	0.0001	0.9%	0.005	71%	0.002	28%
512	4-way	0.006	0.0001	1.1%	0.005	91%	0.000	8%
512	8-way	0.006	0.0001	1.1%	0.005	95%	0.000	4%

Figure B.8 Total miss rate for each size cache and percentage of each according to the three C's. Compulsory misses are independent of cache size, while capacity misses decrease as capacity increases, and conflict misses decrease as associativity increases. Figure B.9 shows the same information graphically. Note that a direct-mapped cache of size *N* has about the same miss rate as a two-way set-associative cache of size *N*/2 up through 128 K. Caches larger than 128 KiB do not prove that rule. Note that the Capacity column is also the fully associative miss rate. Data were collected as in Figure B.4 using LRU replacement.

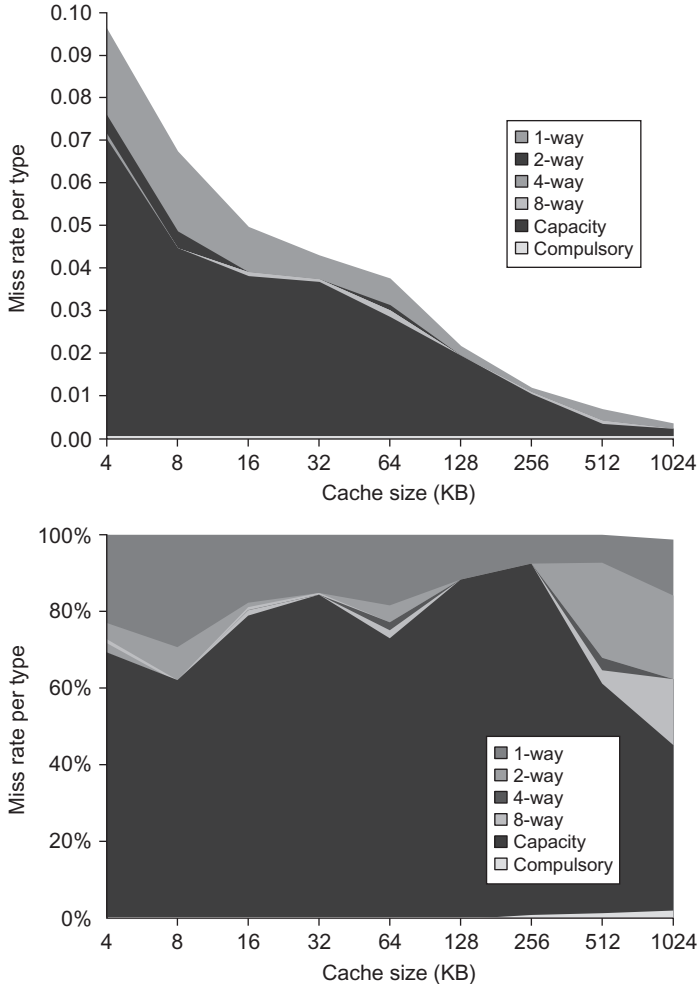

Figure B.9 Total miss rate (top) and distribution of miss rate (bottom) for each size cache according to the three C's for the data in Figure B.8. The top diagram shows the actual data cache miss rates, while the bottom diagram shows the percentage in each category. (*Space allows* the graphs to show one extra cache size than can fit in Figure B.8.)

conflict misses. Full associativity is expensive in hardware, however, and may slow the processor clock rate (see the example on page B-29), leading to lower overall performance.

There is little to be done about capacity except to enlarge the cache. If the upper-level memory is much smaller than what is needed for a program, and a significant percentage of the time is spent moving data between two levels in the

hierarchy, the memory hierarchy is said to *thrash*. Because so many replacements are required, thrashing means the computer runs close to the speed of the lower-level memory, or maybe even slower because of the miss overhead.

Another approach to improving the three C's is to make blocks larger to reduce the number of compulsory misses, but, as we will see shortly, large blocks can increase other kinds of misses.

The three C's give insight into the cause of misses, but this simple model has its limits; it gives you insight into average behavior but may not explain an individual miss. For example, changing cache size changes conflict misses as well as capacity misses, because a larger cache spreads out references to more blocks. Thus, a miss might move from a capacity miss to a conflict miss as cache size changes. Similarly, changing the block size can sometimes reduce capacity misses (in addition to the expected reduction in compusolory misses), as Gupta et al. (2013) show.

Note also that the three C's also ignore replacement policy, because it is difficult to model and because, in general, it is less significant. In specific circumstances the replacement policy can actually lead to anomalous behavior, such as poorer miss rates for larger associativity, which contradicts the three C's model. (Some have proposed using an address trace to determine optimal placement in memory to avoid placement misses from the three C's model; we've not followed that advice here.)

Alas, many of the techniques that reduce miss rates also increase hit time or miss penalty. The desirability of reducing miss rates using the three optimizations must be balanced against the goal of making the whole system fast. This first example shows the importance of a balanced perspective.

First Optimization: Larger Block Size to Reduce Miss Rate

The simplest way to reduce miss rate is to increase the block size. Figure B.10 shows the trade-off of block size versus miss rate for a set of programs and cache sizes. Larger block sizes will reduce also compulsory misses. This reduction occurs because the principle of locality has two components: temporal locality and spatial locality. Larger blocks take advantage of spatial locality.

At the same time, larger blocks increase the miss penalty. Because they reduce the number of blocks in the cache, larger blocks may increase conflict misses and even capacity misses if the cache is small. Clearly, there is little reason to increase the block size to such a size that it *increases* the miss rate. There is also no benefit to reducing miss rate if it increases the average memory access time. The increase in miss penalty may outweigh the decrease in miss rate.

Example Figure B.11 shows the actual miss rates plotted in Figure B.10. Assume the memory system takes 80 clock cycles of overhead and then delivers 16 bytes every 2 clock cycles. Thus, it can supply 16 bytes in 82 clock cycles, 32 bytes in 84 clock cycles, and so on. Which block size has the smallest average memory access time for each cache size in Figure B.11?

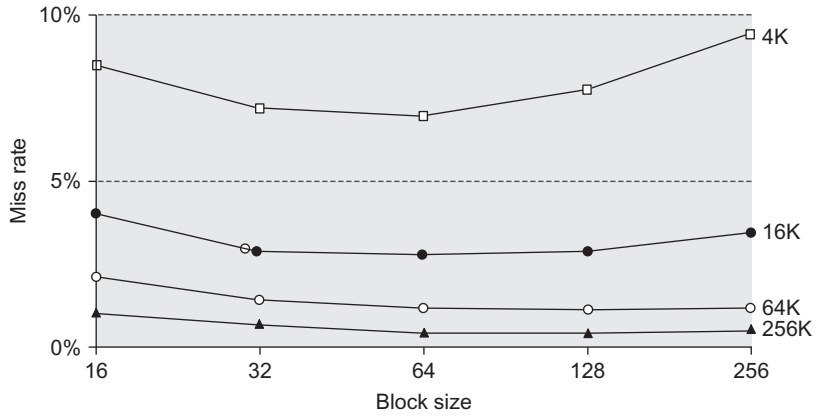

Figure B.10 Miss rate versus block size for five different-sized caches. Note that miss rate actually goes up if the block size is too large relative to the cache size. Each line represents a cache of different size. Figure B.11 shows the data used to plot these lines. Unfortunately, SPEC2000 traces would take too long if block size were included, so these data are based on SPEC92 on a DECstation 5000 (Gee et al. 1993).

Answer Average memory access time is

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

If we assume the hit time is 1 clock cycle independent of block size, then the access time for a 16-byte block in a 4 KiB cache is

$$\text{Average memory access time} = 1 + (8.57\% \times 82) = 8.027 \text{ clock cycles}$$

and for a 256-byte block in a 256 KiB cache the average memory access time is

$$\text{Average memory access time} = 1 + (0.49\% \times 112) = 1.549 \text{ clock cycles}$$

	Cache size			
Block size	**4K**	**16K**	**64K**	**256K**
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Figure B.11 Actual miss rate versus block size for the five different-sized caches in Figure B.10. Note that for a 4 KiB cache, 256-byte blocks have a higher miss rate than 32-byte blocks. In this example, the cache would have to be 256 KiB in order for a 256-byte block to decrease misses.

		Cache size			
Block size	**Miss penalty**	**4K**	**16K**	**64K**	**256K**
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

Figure B.12 Average memory access time versus block size for five different-sized caches in Figure B.10. Block sizes of 32 and 64 bytes dominate. The smallest average time per cache size is boldfaced.

Figure B.12 shows the average memory access time for all block and cache sizes between those two extremes. The boldfaced entries show the fastest block size for a given cache size: 32 bytes for 4 KiB and 64 bytes for the larger caches. These sizes are, in fact, popular block sizes for processor caches today.

As in all of these techniques, the cache designer is trying to minimize both the miss rate and the miss penalty. The selection of block size depends on both the latency and bandwidth of the lower-level memory. High latency and high bandwidth encourage large block size because the cache gets many more bytes per miss for a small increase in miss penalty. Conversely, low latency and low bandwidth encourage smaller block sizes because there is little time saved from a larger block. For example, twice the miss penalty of a small block may be close to the penalty of a block twice the size. The larger number of small blocks may also reduce conflict misses. Note that Figures B.10 and B.12 show the difference between selecting a block size based on minimizing miss rate versus minimizing average memory access time.

After seeing the positive and negative impact of larger block size on compulsory and capacity misses, the next two subsections look at the potential of higher capacity and higher associativity.

Second Optimization: Larger Caches to Reduce Miss Rate

The obvious way to reduce capacity misses in Figures B.8 and B.9 is to increase capacity of the cache. The obvious drawback is potentially longer hit time and higher cost and power. This technique has been especially popular in off-chip caches.

Third Optimization: Higher Associativity to Reduce Miss Rate

Figures B.8 and B.9 show how miss rates improve with higher associativity. There are two general rules of thumb that can be gleaned from these figures. The first is

that eight-way set associative is for practical purposes as effective in reducing misses for these sized caches as fully associative. You can see the difference by comparing the eight-way entries to the capacity miss column in Figure B.8, because capacity misses are calculated using fully associative caches.

The second observation, called the *2:1 cache rule of thumb*, is that a direct-mapped cache of size *N* has about the same miss rate as a two-way set associative cache of size *N*/2. This held in three C's figures for cache sizes less than 128 KiB.

Like many of these examples, improving one aspect of the average memory access time comes at the expense of another. Increasing block size reduces miss rate while increasing miss penalty, and greater associativity can come at the cost of increased hit time. Hence, the pressure of a fast processor clock cycle encourages simple cache designs, but the increasing miss penalty rewards associativity, as the following example suggests.

Example Assume that higher associativity would increase the clock cycle time as listed as follows:

$$\text{Clock cycle time}_{\text{2-way}} = 1.36 \times \text{Clock cycle time}_{\text{1-way}}$$
$$\text{Clock cycle time}_{\text{4-way}} = 1.44 \times \text{Clock cycle time}_{\text{1-way}}$$
$$\text{Clock cycle time}_{\text{8-way}} = 1.52 \times \text{Clock cycle time}_{\text{1-way}}$$

Assume that the hit time is 1 clock cycle, that the miss penalty for the direct-mapped case is 25 clock cycles to a level 2 cache (see next subsection) that never misses, and that the miss penalty need not be rounded to an integral number of clock cycles. Using Figure B.8 for miss rates, for which cache sizes are each of these three statements true?

$$\text{Average memory access time}_{\text{8-way}} < \text{Average memory access time}_{\text{4-way}}$$
$$\text{Average memory access time}_{\text{4-way}} < \text{Average memory access time}_{\text{2-way}}$$
$$\text{Average memory access time}_{\text{2-way}} < \text{Average memory access time}_{\text{1-way}}$$

Answer Average memory access time for each associativity is

$$\begin{aligned}\text{Average memory access time}_{\text{8-way}} &= \text{Hit time}_{\text{8-way}} + \text{Miss rate}_{\text{8-way}} \times \text{Miss penalty}_{\text{8-way}}\\ &= 1.52 + \text{Miss rate}_{\text{8-way}} \times 25\\ \text{Average memory access time}_{\text{4-way}} &= 1.44 + \text{Miss rate}_{\text{4-way}} \times 25\\ \text{Average memory access time}_{\text{2-way}} &= 1.36 + \text{Miss rate}_{\text{2-way}} \times 25\\ \text{Average memory access time}_{\text{1-way}} &= 1.00 + \text{Miss rate}_{\text{1-way}} \times 25\end{aligned}$$

The miss penalty is the same time in each case, so we leave it as 25 clock cycles. For example, the average memory access time for a 4 KiB direct-mapped cache is

$$\text{Average memory access time}_{\text{1-way}} = 1.00 + (0.098 \times 25) = 3.44$$

and the time for a 512 KiB, eight-way set associative cache is

$$\text{Average memory access time}_{\text{8-way}} = 1.52 + (0.006 \times 25) = 1.66$$

Using these formulas and the miss rates from Figure B.8, Figure B.13 shows the average memory access time for each cache and associativity. The figure shows

	Associativity			
Cache size (KiB)	1-way	2-way	4-way	8-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Figure B.13 Average memory access time using miss rates in Figure B.8 for parameters in the example. *Boldface* type means that this time is higher than the number to the left, that is, higher associativity *increases* average memory access time.

that the formulas in this example hold for caches less than or equal to 8 KiB for up to four-way associativity. Starting with 16 KiB, the greater hit time of larger associativity outweighs the time saved due to the reduction in misses.

Note that we did not account for the slower clock rate on the rest of the program in this example, thereby understating the advantage of direct-mapped cache.

Fourth Optimization: Multilevel Caches to Reduce Miss Penalty

Reducing cache misses had been the traditional focus of cache research, but the cache performance formula assures us that improvements in miss penalty can be just as beneficial as improvements in miss rate. Moreover, Figure 2.2 on page 80 shows that technology trends have improved the speed of processors faster than DRAMs, making the relative cost of miss penalties increase over time.

This performance gap between processors and memory leads the architect to this question: Should I make the cache faster to keep pace with the speed of processors, or make the cache larger to overcome the widening gap between the processor and main memory?

One answer is, do both. Adding another level of cache between the original cache and memory simplifies the decision. The first-level cache can be small enough to match the clock cycle time of the fast processor. Yet, the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

Although the concept of adding another level in the hierarchy is straightforward, it complicates performance analysis. Definitions for a second level of cache are not always straightforward. Let's start with the definition of *average memory access time* for a two-level cache. Using the subscripts L1 and L2 to refer, respectively, to a first-level and a second-level cache, the original formula is

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times \text{Miss penalty}_{L1}$$

and

$$\text{Miss penalty}_{L1} = \text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}$$

so

$$\text{Average memory access time} = \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2})$$

In this formula, the second-level miss rate is measured on the leftovers from the first-level cache. To avoid ambiguity, these terms are adopted here for a two-level cache system:

- *Local miss rate*—This rate is simply the number of misses in a cache divided by the total number of memory accesses to this cache. As you would expect, for the first-level cache it is equal to Miss rate_{L1}, and for the second-level cache it is Miss rate_{L2}.
- *Global miss rate*—The number of misses in the cache divided by the total number of memory accesses generated by the processor. Using the terms above, the global miss rate for the first-level cache is still just Miss rate_{L1}, but for the second-level cache it is $\text{Miss rate}_{L1} \times \text{Miss rate}_{L2}$.

This local miss rate is large for second-level caches because the first-level cache skims the cream of the memory accesses. This is why the global miss rate is the more useful measure: It indicates what fraction of the memory accesses that leave the processor go all the way to memory.

Here is a place where the misses per instruction metric shines. Instead of confusion about local or global miss rates, we just expand memory stalls per instruction to add the impact of a second-level cache.

$$\text{Average memory stalls per instruction} = \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \times \text{Miss penalty}_{L2}$$

Example Suppose that in 1000 memory references there are 40 misses in the first-level cache and 20 misses in the second-level cache. What are the various miss rates? Assume the miss penalty from the L2 cache to memory is 200 clock cycles, the hit time of the L2 cache is 10 clock cycles, the hit time of L1 is 1 clock cycle, and there are 1.5 memory references per instruction. What is the average memory access time and average stall cycles per instruction? Ignore the impact of writes.

Answer The miss rate (either local or global) for the first-level cache is 40/1000 or 4%. The local miss rate for the second-level cache is 20/40 or 50%. The global miss rate of the second-level cache is 20/1000 or 2%. Then

$$\begin{aligned}\text{Average memory access time} &= \text{Hit time}_{L1} + \text{Miss rate}_{L1} \times (\text{Hit time}_{L2} + \text{Miss rate}_{L2} \times \text{Miss penalty}_{L2}) \\ &= 1 + 4\% \times (10 + 50\% \times 200) = 1 + 4\% \times 110 = 5.4 \text{ clock cycles}\end{aligned}$$

To see how many misses we get per instruction, we divide 1000 memory references by 1.5 memory references per instruction, which yields 667 instructions. Thus, we need to multiply the misses by 1.5 to get the number of misses per 1000 instructions. We have 40×1.5 or 60 L1 misses, and 20×1.5 or 30 L2 misses, per 1000 instructions. For average memory stalls per instruction, assuming the misses are distributed uniformly between instructions and data:

$$\begin{aligned}\text{Average memory stalls per instruction} &= \text{Misses per instruction}_{L1} \times \text{Hit time}_{L2} + \text{Misses per instruction}_{L2} \\ &\quad \times \text{Miss penalty}_{L2} \\ &= (60/1000) \times 10 + (30/1000) \times 200 \\ &= 0.060 \times 10 + 0.030 \times 200 = 6.6 \text{ clock cycles}\end{aligned}$$

If we subtract the L1 hit time from the average memory access time (AMAT) and then multiply by the average number of memory references per instruction, we get the same average memory stalls per instruction:

$$(5.4 - 1.0) \times 1.5 = 4.4 \times 1.5 = 6.6 \text{ clock cycles}$$

As this example shows, there may be less confusion with multilevel caches when calculating using misses per instruction versus miss rates.

Note that these formulas are for combined reads and writes, assuming a write-back first-level cache. Obviously, a write-through first-level cache will send *all* writes to the second level, not just the misses, and a write buffer might be used.

Figures B.14 and B.15 show how miss rates and relative execution time change with the size of a second-level cache for one design. From these figures we can gain two insights. The first is that the global cache miss rate is very similar to the single cache miss rate of the second-level cache, provided that the second-level cache is much larger than the first-level cache. Hence, our intuition and knowledge about the first-level caches apply. The second insight is that the local cache miss rate is *not* a good measure of secondary caches; it is a function of the miss rate of the first-level cache, and hence can vary by changing the first-level cache. Thus, the global cache miss rate should be used when evaluating second-level caches.

With these definitions in place, we can consider the parameters of second-level caches. The foremost difference between the two levels is that the speed of the first-level cache affects the clock rate of the processor, while the speed of the second-level cache only affects the miss penalty of the first-level cache. Thus, we can consider many alternatives in the second-level cache that would be ill chosen for the first-level cache. There are two major questions for the design of the second-level cache: Will it lower the average memory access time portion of the CPI, and how much does it cost?

The initial decision is the size of a second-level cache. Since everything in the first-level cache is likely to be in the second-level cache, the second-level cache should be much bigger than the first. If second-level caches are just a little bigger, the local miss rate will be high. This observation inspires the design of huge second-level caches—the size of main memory in older computers!

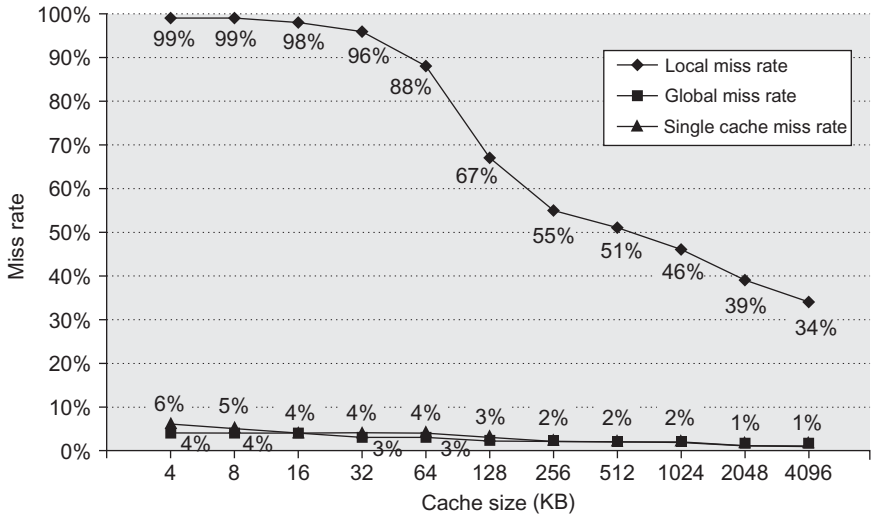

Figure B.14 Miss rates versus cache size for multilevel caches. Second-level caches *smaller* than the sum of the two 64 KiB first-level caches make little sense, as reflected in the high miss rates. After 256 KiB the single cache is within 10% of the global miss rates. The miss rate of a single-level cache versus size is plotted against the local miss rate and global miss rate of a second-level cache using a 32 KiB first-level cache. The L2 caches (unified) were two-way set associative with replacement. Each had split L1 instruction and data caches that were 64 KiB two-way set associative with LRU replacement. The block size for both L1 and L2 caches was 64 bytes. Data were collected as in Figure B.4.

One question is whether set associativity makes more sense for second-level caches.

Example Given the following data, what is the impact of second-level cache associativity on its miss penalty?

- Hit time$_{L2}$ for direct mapped = 10 clock cycles.
- Two-way set associativity increases hit time by 0.1 clock cycle to 10.1 clock cycles.
- Local miss rate$_{L2}$ for direct mapped = 25%.
- Local miss rate$_{L2}$ for two-way set associative = 20%.
- Miss penalty$_{L2}$ = 200 clock cycles.

Answer For a direct-mapped second-level cache, the first-level cache miss penalty is

$$\text{Miss penalty}_{\text{1-way L2}} = 10 + 25\% \times 200 = 60.0 \text{ clock cycles}$$

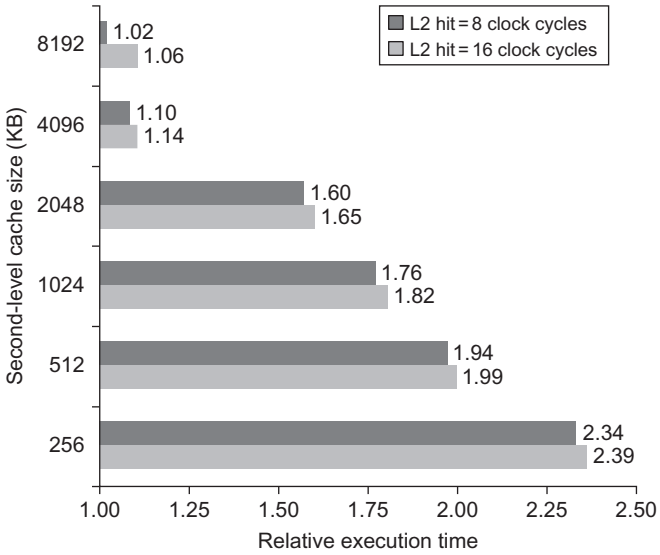

Figure B.15 Relative execution time by second-level cache size. The two bars are for different clock cycles for an L2 cache hit. The reference execution time of 1.00 is for an 8192 KiB second-level cache with a 1-clock-cycle latency on a second-level hit. These data were collected the same way as in Figure B.14, using a simulator to imitate the Alpha 21264.

Adding the cost of associativity increases the hit cost only 0.1 clock cycle, making the new first-level cache miss penalty:

$$\text{Miss penalty}_{\text{2-way L2}} = 10.1 + 20\% \times 200 = 50.1 \text{ clock cycles}$$

In reality, second-level caches are almost always synchronized with the first-level cache and processor. Accordingly, the second-level hit time must be an integral number of clock cycles. If we are lucky, we shave the second-level hit time to 10 cycles; if not, we round up to 11 cycles. Either choice is an improvement over the direct-mapped second-level cache:

$$\text{Miss penalty}_{\text{2-way L2}} = 10 + 20\% \times 200 = 50.0 \text{ clock cycles}$$
$$\text{Miss penalty}_{\text{2-way L2}} = 11 + 20\% \times 200 = 51.0 \text{ clock cycles}$$

Now we can reduce the miss penalty by reducing the *miss rate* of the second-level caches.

Another consideration concerns whether data in the first-level cache are in the second-level cache. *Multilevel inclusion* is the natural policy for memory hierarchies: L1 data are always present in L2. Inclusion is desirable because consistency between I/O and caches (or among caches in a multiprocessor) can be determined just by checking the second-level cache.

One drawback to inclusion is that measurements can suggest smaller blocks for the smaller first-level cache and larger blocks for the larger second-level cache. For example, the Pentium 4 has 64-byte blocks in its L1 caches and 128-byte blocks in its L2 cache. Inclusion can still be maintained with more work on a second-level miss. The second-level cache must invalidate all first-level blocks that map onto the second-level block to be replaced, causing a slightly higher first-level miss rate. To avoid such problems, many cache designers keep the block size the same in all levels of caches.

However, what if the designer can only afford an L2 cache that is slightly bigger than the L1 cache? Should a significant portion of its space be used as a redundant copy of the L1 cache? In such cases a sensible opposite policy is *multilevel exclusion:* L1 data are *never* found in an L2 cache. Typically, with exclusion a cache miss in L1 results in a swap of blocks between L1 and L2 instead of a replacement of an L1 block with an L2 block. This policy prevents wasting space in the L2 cache. For example, the AMD Opteron chip obeys the exclusion property using two 64 KiB L1 caches and 1 MiB L2 cache.

As these issues illustrate, although a novice might design the first- and second-level caches independently, the designer of the first-level cache has a simpler job given a compatible second-level cache. It is less of a gamble to use a write through, for example, if there is a write-back cache at the next level to act as a backstop for repeated writes and it uses multilevel inclusion.

The essence of all cache designs is balancing fast hits and few misses. For second-level caches, there are far fewer hits than in the first-level cache, so the emphasis shifts to fewer misses. This insight leads to much larger caches and techniques to lower the miss rate, such as higher associativity and larger blocks.

Fifth Optimization: Giving Priority to Read Misses over Writes to Reduce Miss Penalty

This optimization serves reads before writes have been completed. We start with looking at the complexities of a write buffer.

With a write-through cache the most important improvement is a write buffer of the proper size. Write buffers, however, do complicate memory accesses because they might hold the updated value of a location needed on a read miss.

Example Look at this code sequence:

```
sd x3, 512(x0);M[512] ¬ R3 (cache index 0)
ld x1, 1024(x0);x1 ¬ M[1024](cache index 0)
ld x2, 512(x0);x2 ¬ M[512] (cache index 0)
```

Assume a direct-mapped, write-through cache that maps 512 and 1024 to the same block, and a four-word write buffer that is not checked on a read miss. Will the value in x2 always be equal to the value in x3?

Answer Using the terminology from Chapter 2, this is a read-after-write data hazard in memory. Let's follow a cache access to see the danger. The data in x3 are placed into the write buffer after the store. The following load uses the same cache index and is therefore a miss. The second load instruction tries to put the value in location 512 into register x2; this also results in a miss. If the write buffer hasn't completed writing to location 512 in memory, the read of location 512 will put the old, wrong value into the cache block, and then into x2. Without proper precautions, x3x1 would not be equal to x2!

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

The cost of writes by the processor in a write-back cache can also be reduced. Suppose a read miss will replace a dirty memory block. Instead of writing the dirty block to memory, and then reading memory, we could copy the dirty block to a buffer, then read memory, and *then* write memory. This way the processor read, for which the processor is probably waiting, will finish sooner. Similar to the previous situation, if a read miss occurs, the processor can either stall until the buffer is empty or check the addresses of the words in the buffer for conflicts.

Now that we have five optimizations that reduce cache miss penalties or miss rates, it is time to look at reducing the final component of average memory access time. Hit time is critical because it can affect the clock rate of the processor; in many processors today the cache access time limits the clock cycle rate, even for processors that take multiple clock cycles to access the cache. Hence, a fast hit time is multiplied in importance beyond the average memory access time formula because it helps everything.

Sixth Optimization: Avoiding Address Translation During Indexing of the Cache to Reduce Hit Time

Even a small and simple cache must cope with the translation of a virtual address from the processor to a physical address to access memory. As described in Section B.4, processors treat main memory as just another level of the memory hierarchy, and thus the address of the virtual memory that exists on disk must be mapped onto the main memory.

The guideline of making the common case fast suggests that we use virtual addresses for the cache, because hits are much more common than misses. Such caches are termed *virtual caches*, with *physical cache* used to identify the traditional cache that uses physical addresses. As we will shortly see, it is important to distinguish two tasks: indexing the cache and comparing addresses. Thus, the issues are whether a virtual or physical address is used to index the cache and

whether a virtual or physical address is used in the tag comparison. Full virtual addressing for both indices and tags eliminates address translation time from a cache hit. Then why doesn't everyone build virtually addressed caches?

One reason is protection. Page-level protection is checked as part of the virtual to physical address translation, and it must be enforced no matter what. One solution is to copy the protection information from the TLB on a miss, add a field to hold it, and check it on every access to the virtually addressed cache.

Another reason is that every time a process is switched, the virtual addresses refer to different physical addresses, requiring the cache to be flushed. Figure B.16 shows the impact on miss rates of this flushing. One solution is to increase the width of the cache address tag with a *process-identifier tag* (PID). If the operating system assigns these tags to processes, it only need flush the cache when a PID is

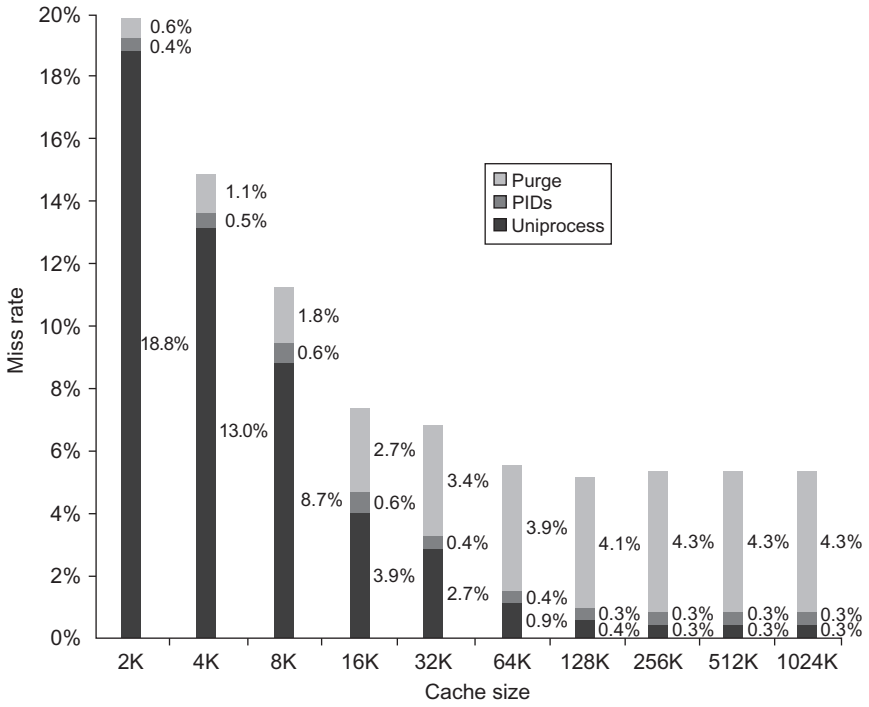

Figure B.16 Miss rate versus virtually addressed cache size of a program measured three ways: without process switches (uniprocess), with process switches using a process-identifier tag (PID), and with process switches but without PIDs (purge). PIDs increase the uniprocess absolute miss rate by 0.3%–0.6% and save 0.6%–4.3% over purging. Agarwal (1987) collected these statistics for the Ultrix operating system running on a VAX, assuming direct-mapped caches with a block size of 16 bytes. Note that the miss rate goes up from 128 to 256 K. Such nonintuitive behavior can occur in caches because changing size changes the mapping of memory blocks onto cache blocks, which can change the conflict miss rate.

recycled; that is, the PID distinguishes whether or not the data in the cache are for this program. Figure B.16 shows the improvement in miss rates by using PIDs to avoid cache flushes.

A third reason why virtual caches are not more popular is that operating systems and user programs may use two different virtual addresses for the same physical address. These duplicate addresses, called *synonyms* or *aliases*, could result in two copies of the same data in a virtual cache; if one is modified, the other will have the wrong value. With a physical cache this wouldn't happen, because the accesses would first be translated to the same physical cache block.

Hardware solutions to the synonym problem, called *antialiasing*, guarantee every cache block a unique physical address. For example, the AMD Opteron uses a 64 KiB instruction cache with a 4 KiB page and two-way set associativity; hence, the hardware must handle aliases involved with the three virtual address bits in the set index. It avoids aliases by simply checking all eight possible locations on a miss—two blocks in each of four sets—to be sure that none matches the physical address of the data being fetched. If one is found, it is invalidated, so when the new data are loaded into the cache their physical address is guaranteed to be unique.

Software can make this problem much easier by forcing aliases to share some address bits. An older version of UNIX from Sun Microsystems, for example, required all aliases to be identical in the last 18 bits of their addresses; this restriction is called *page coloring*. Note that page coloring is simply set associative mapping applied to virtual memory: the 4 KiB (2^{12}) pages are mapped using 64 (2^6) sets to ensure that the physical and virtual addresses match in the last 18 bits. This restriction means a direct-mapped cache that is 2^{18} (256 K) bytes or smaller can never have duplicate physical addresses for blocks. From the perspective of the cache, page coloring effectively increases the page offset, as software guarantees that the last few bits of the virtual and physical page address are identical.

The final area of concern with virtual addresses is I/O. I/O typically uses physical addresses and thus would require mapping to virtual addresses to interact with a virtual cache. (The impact of I/O on caches is further discussed in Appendix D.)

One alternative to get the best of both virtual and physical caches is to use part of the page offset—the part that is identical in both virtual and physical addresses—to index the cache. At the same time as the cache is being read using that index, the virtual part of the address is translated, and the tag match uses physical addresses.

This alternative allows the cache read to begin immediately, and yet the tag comparison is still with physical addresses. The limitation of this *virtually indexed, physically tagged* alternative is that a direct-mapped cache can be no bigger than the page size. For example, in the data cache in Figure B.5 on page B-13, the index is 9 bits and the cache block offset is 6 bits. To use this trick, the virtual page size would have to be at least $2^{(9+6)}$ bytes or 32 KiB. If not, a portion of the index must be translated from virtual to physical address. Figure B.17 shows the organization of the caches, translation lookaside buffers (TLBs), and virtual memory when this technique is used.

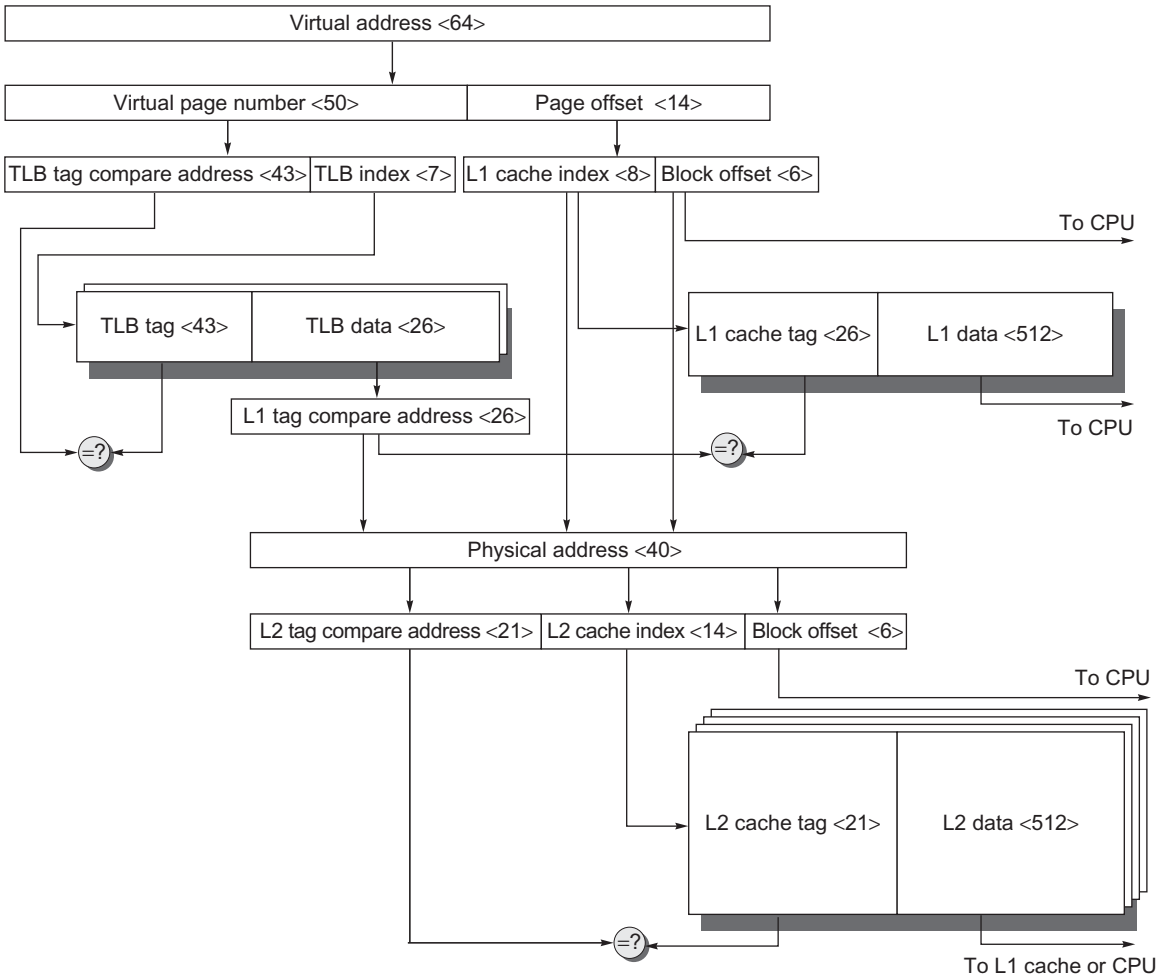

Figure B.17 The overall picture of a hypothetical memory hierarchy going from virtual address to L2 cache access. The page size is 16 KiB. The TLB is two-way set associative with 256 entries. The L1 cache is a direct-mapped 16 KiB, and the L2 cache is a four-way set associative with a total of 4 MiB. Both use 64-byte blocks. The virtual address is 64 bits and the physical address is 40 bits.

Associativity can keep the index in the physical part of the address and yet still support a large cache. Recall that the size of the index is controlled by this formula:

$$2^{\text{Index}} = \frac{\text{Cache size}}{\text{Block size} \times \text{Set associativity}}$$

For example, doubling associativity and doubling the cache size does not change the size of the index. The IBM 3033 cache, as an extreme example, is 16-way set associative, even though studies show there is little benefit to miss rates above

Technique	Hit time	Miss penalty	Miss rate	Hardware complexity	Comment
Larger block size		–	+	0	Trivial; Pentium 4L2 uses 128 bytes
Larger cache size	–		+	1	Widely used, especially for L2 caches
Higher associativity	–		+	1	Widely used
Multilevel caches		+		2	Costly hardware; harder if L1 block size ≠ L2 block size; widely used
Read priority over writes		+		1	Widely used
Avoiding address translation during cache indexing	+			1	Widely used

Figure B.18 Summary of basic cache optimizations showing impact on cache performance and complexity for the techniques in this appendix. Generally a technique helps only one factor. + means that the technique improves the factor, – means it hurts that factor, and blank means it has no impact. The complexity measure is subjective, with 0 being the easiest and 3 being a challenge.

8-way set associativity. This high associativity allows a 64 KiB cache to be addressed with a physical index, despite the handicap of 4 KiB pages in the IBM architecture.

Summary of Basic Cache Optimization

The techniques in this section to improve miss rate, miss penalty, and hit time generally impact the other components of the average memory access equation as well as the complexity of the memory hierarchy. Figure B.18 summarizes these techniques and estimates the impact on complexity, with + meaning that the technique improves the factor, – meaning it hurts that factor, and blank meaning it has no impact. No optimization in this figure helps more than one category.

B.4 Virtual Memory

… a system has been devised to make the core drum combination appear to the programmer as a single level store, the requisite transfers taking place automatically.

Kilburn et al. (1962)

At any instant in time computers are running multiple processes, each with its own address space. (Processes are described in the next section.) It would be too expensive to dedicate a full address space worth of memory for each process, especially because many processes use only a small part of their address space. Hence, there must be a means of sharing a smaller amount of physical memory among many processes.