opportunities. It is likely that the use of instruction- and thread-level parallelism will be a more important tool in hiding whatever memory delays are encountered in modern multilevel cache systems.

One idea that periodically arises is the use of programmer-controlled scratchpad or other high-speed visible memories, which we will see are used in GPUs. Such ideas have never made the mainstream in general-purpose processors for several reasons: First, they break the memory model by introducing address spaces with different behavior. Second, unlike compiler-based or programmer-based cache optimizations (such as prefetching), memory transformations with scratchpads must completely handle the remapping from main memory address space to the scratchpad address space. This makes such transformations more difficult and limited in applicability. In GPUs (see Chapter 4), where local scratchpad memories are heavily used, the burden for managing them currently falls on the programmer. For domain-specific software systems that can use such memories, the performance gains are very significant. It is likely that HBM technologies will thus be used for caching in large, general-purpose computers and quite possibility as the main working memories in graphics and similar systems. As domain-specific architectures become more important in overcoming the limitations arising from the end of Dennard's Law and the slowdown in Moore's Law (see Chapter 7), scratchpad memories and vector-like register sets are likely to see more use.

The implications of the end of Dennard's Law affect both DRAM and processor technology. Thus, rather than a widening gulf between processors and main memory, we are likely to see a slowdown in both technologies, leading to slower overall growth rates in performance. New innovations in computer architecture and in related software that together increase performance and efficiency will be key to continuing the performance improvements seen over the past 50 years.

2.9 Historical Perspectives and References

In Section M.3 (available online) we examine the history of caches, virtual memory, and virtual machines. IBM plays a prominent role in the history of all three. References for further reading are included.

Case Studies and Exercises by Norman P. Jouppi, Rajeev Balasubramonian, Naveen Muralimanohar, and Sheng Li

Case Study 1: Optimizing Cache Performance via Advanced Techniques

Concepts illustrated by this case study

- Nonblocking Caches
- Compiler Optimizations for Caches
- Software and Hardware Prefetching
- Calculating Impact of Cache Performance on More Complex Processors

The transpose of a matrix interchanges its rows and columns; this concept is illustrated here:

$$\begin{bmatrix} A11 & A12 & A13 & A14 \\ A21 & A22 & A23 & A24 \\ A31 & A32 & A33 & A34 \\ A41 & A42 & A43 & A44 \end{bmatrix} \Rightarrow \begin{bmatrix} A11 & A21 & A31 & A41 \\ A12 & A22 & A32 & A42 \\ A13 & A23 & A33 & A43 \\ A14 & A24 & A34 & A44 \end{bmatrix}$$

Here is a simple C loop to show the transpose:

```
for (i = 0; i < 3; i++) {
 for (j = 0; j < 3; j++) {
 output[j][i] = input[i][j];
 }
}
```

Assume that both the input and output matrices are stored in the row major order (*row major order* means that the row index changes fastest). Assume that you are executing a 256·256 double-precision transpose on a processor with a 16 KB fully associative (don't worry about cache conflicts) least recently used (LRU) replacement L1 data cache with 64-byte blocks. Assume that the L1 cache misses or prefetches require 16 cycles and always hit in the L2 cache, and that the L2 cache can process a request every 2 processor cycles. Assume that each iteration of the preceding inner loop requires 4 cycles if the data are present in the L1 cache. Assume that the cache has a write-allocate fetch-on-write policy for write misses. Unrealistically, assume that writing back dirty cache blocks requires 0 cycles.

2.1 [10/15/15/12/20] <2.3> For the preceding simple implementation, this execution order would be nonideal for the input matrix; however, applying a loop interchange optimization would create a nonideal order for the output matrix. Because loop interchange is not sufficient to improve its performance, it must be blocked instead.

a. [10] <2.3> What should be the minimum size of the cache to take advantage of blocked execution?

b. [15] <2.3> How do the relative number of misses in the blocked and unblocked versions compare in the preceding minimum-sized cache?

c. [15] <2.3> Write code to perform a transpose with a block size parameter B that uses $B \cdot B$ blocks.

d. [12] <2.3> What is the minimum associativity required of the L1 cache for consistent performance independent of both arrays' position in memory?

e. [20] <2.3> Try out blocked and nonblocked 256·256 matrix transpositions on a computer. How closely do the results match your expectations based on what you know about the computer's memory system? Explain any discrepancies if possible.

2.2 [10] <2.3> Assume you are designing a hardware prefetcher for the preceding *unblocked* matrix transposition code. The simplest type of hardware prefetcher only prefetches sequential cache blocks after a miss. More complicated "nonunit stride" hardware prefetchers can analyze a miss reference stream and detect and prefetch nonunit strides. In contrast, software prefetching can determine nonunit strides as easily as it can determine unit strides. Assume prefetches write directly into the cache and that there is no "pollution" (overwriting data that must be used before the data that are prefetched). For best performance given a nonunit stride prefetcher, in the steady state of the inner loop, how many prefetches must be outstanding at a given time?

2.3 [15/20] <2.3> With software prefetching, it is important to be careful to have the prefetches occur in time for use but also to minimize the number of outstanding prefetches to live within the capabilities of the microarchitecture and minimize cache pollution. This is complicated by the fact that different processors have different capabilities and limitations.

a. [15] <2.3> Create a blocked version of the matrix transpose with software prefetching.

b. [20] <2.3> Estimate and compare the performance of the blocked and unblocked transpose codes both with and without software prefetching.

Case Study 2: Putting It All Together: Highly Parallel Memory Systems

Concept illustrated by this case study

■ Cross-Cutting Issues: The Design of Memory Hierarchies

The program in Figure 2.32 can be used to evaluate the behavior of a memory system. The key is having accurate timing and then having the program stride through memory to invoke different levels of the hierarchy. Figure 2.32 shows the code in C. The first part is a procedure that uses a standard utility to get an accurate measure of the user CPU time; this procedure may have to be changed to work on some systems. The second part is a nested loop to read and write memory at different strides and cache sizes. To get accurate cache timing, this code is repeated many times. The third part times the nested loop overhead only so that it can be subtracted from overall measured times to see how long the accesses were. The results are output in `.csv` file format to facilitate importing into spreadsheets. You may need to change `CACHE_MAX` depending on the question you are answering and the size of memory on the system you are measuring. Running the program in single-user mode or at least without other active applications will give more consistent results. The code in Figure 2.32 was derived from a program written by Andrea Dusseau at the University of California-Berkeley and was based on a detailed description found in Saavedra-Barrera (1992). It has been modified to fix a number of issues with more modern machines and to run under Microsoft

```
#include "stdafx.h"
#include <stdio.h>
#include <time.h>
#define ARRAY_MIN (1024) /* 1/4 smallest cache */
#define ARRAY_MAX (4096*4096) /* 1/4 largest cache */
int x[ARRAY_MAX]; /* array going to stride through */

double get_seconds() { /* routine to read time in seconds */
   __time64_t ltime;
   _time64( &ltime );
   return (double) ltime;
}
int label(int i) {/* generate text labels */
   if (i<1e3) printf("%1dB,",i);
   else if (i<1e6) printf("%1dK,",i/1024);
   else if (i<1e9) printf("%1dM,",i/1048576);
   else printf("%1dG,",i/1073741824);
   return 0;
}
int _tmain(int argc, _TCHAR* argv[]) {
int register nextstep, i, index, stride;
int csize;
double steps, tsteps;
double loadtime, lastsec, sec0, sec1, sec; /* timing variables */

/* Initialize output */
printf(" ,");
for (stride=1; stride <= ARRAY_MAX/2; stride=stride*2)
   label(stride*sizeof(int));
printf("\n");

/* Main loop for each configuration */
for (csize=ARRAY_MIN; csize <= ARRAY_MAX; csize=csize*2) {
   label(csize*sizeof(int)); /* print cache size this loop */
   for (stride=1; stride <= csize/2; stride=stride*2) {

      /* Lay out path of memory references in array */
      for (index=0; index < csize; index=index+stride)
         x[index] = index + stride; /* pointer to next */
      x[index-stride] = 0; /* loop back to beginning */

      /* Wait for timer to roll over */
      lastsec = get_seconds();
       sec0 = get_seconds(); while (sec0 == lastsec);

      /* Walk through path in array for twenty seconds */
      /* This gives 5% accuracy with second resolution */
      steps = 0.0; /* number of steps taken */
      nextstep = 0; /* start at beginning of path */
      sec0 = get_seconds(); /* start timer */
         { /* repeat until collect 20 seconds */
            (i=stride;i!=0;i=i-1) { /* keep samples same */
               nextstep = 0;
               do nextstep = x[nextstep]; /* dependency */
               while (nextstep != 0);
         }
         steps = steps + 1.0; /* count loop iterations */
         sec1 = get_seconds(); /* end timer */
      } while ((sec1 - sec0) < 20.0); /* collect 20 seconds */
      sec = sec1 - sec0;

      /* Repeat empty loop to loop subtract overhead */
      tsteps = 0.0; /* used to match no. while iterations */
      sec0 = get_seconds(); /* start timer */
         { /* repeat until same no. iterations as above */
            (i=stride;i!=0;i=i-1) { /* keep samples same */
               index = 0;
               do index = index + stride;
               while (index < csize);
         }
         tsteps = tsteps + 1.0;
         sec1 = get_seconds(); /* - overhead */
      } while (tsteps<steps); /* until = no. iterations */
      sec = sec - (sec1 - sec0);
      loadtime = (sec*1e9)/(steps*csize);
      /* write out results in .csv format for Excel */
      printf("%4.1f,", (loadtime<0.1) ? 0.1 : loadtime);
     }; /* end of inner for loop */
     printf("\n");
   }; /* end of outer for loop */
   return 0;
}
```

Figure 2.32 C program for evaluating memory system.

Visual C++. It can be downloaded from http://www.hpl.hp.com/research/cacti/aca_ch2_cs2.c.

The preceding program assumes that program addresses track physical addresses, which is true on the few machines that use virtually addressed caches, such as the Alpha 21264. In general, virtual addresses tend to follow physical addresses shortly after rebooting, so you may need to reboot the machine in order to get smooth lines in your results. To answer the following questions, assume that the sizes of all components of the memory hierarchy are powers of 2. Assume that the size of the page is much larger than the size of a block in a second-level cache (if there is one) and that the size of a second-level cache block is greater than or equal to the size of a block in a first-level cache. An example of the output of the program is plotted in Figure 2.33; the key lists the size of the array that is exercised.

2.4 [12/12/12/10/12] <2.6> Using the sample program results in Figure 2.33:

a. [12] <2.6> What are the overall size and block size of the second-level cache?

b. [12] <2.6> What is the miss penalty of the second-level cache?

c. [12] <2.6> What is the associativity of the second-level cache?

d. [10] <2.6> What is the size of the main memory?

e. [12] <2.6> What is the paging time if the page size is 4 KB?

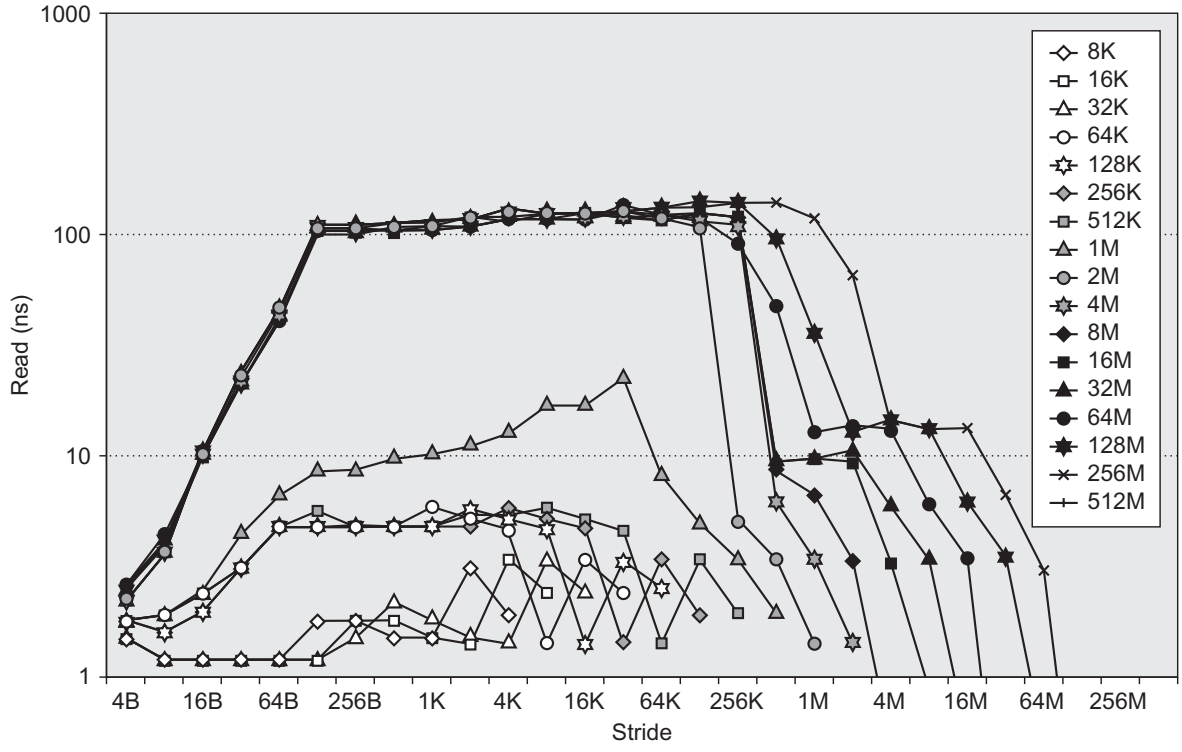

Figure 2.33 Sample results from program in Figure 2.32.

2.5 [12/15/15/20] <2.6> If necessary, modify the code in Figure 2.32 to measure the following system characteristics. Plot the experimental results with elapsed time on the *y*-axis and the memory stride on the *x*-axis. Use logarithmic scales for both axes, and draw a line for each cache size.

a. [12] <2.6> What is the system page size?

b. [15] <2.6> How many entries are there in the TLB?

c. [15] <2.6> What is the miss penalty for the TLB?

d. [20] <2.6> What is the associativity of the TLB?

2.6 [20/20] <2.6> In multiprocessor memory systems, lower levels of the memory hierarchy may not be able to be saturated by a single processor but should be able to be saturated by multiple processors working together. Modify the code in Figure 2.32, and run multiple copies at the same time. Can you determine:

a. [20] <2.6> How many actual processors are in your computer system and how many system processors are just additional multithreaded contexts?

b. [20] <2.6> How many memory controllers does your system have?

2.7 [20] <2.6> Can you think of a way to test some of the characteristics of an instruction cache using a program? *Hint*: The compiler may generate a large number of nonobvious instructions from a piece of code. Try to use simple arithmetic instructions of known length in your instruction set architecture (ISA).

Case Study 3: Studying the Impact of Various Memory System Organizations

Concepts illustrated by this case study

- DDR3 memory systems
- Impact of ranks, banks, row buffers on performance and power
- DRAM timing parameters

A processor chip typically supports a few DDR3 or DDR4 memory channels. We will focus on a single memory channel in this case study and explore how its performance and power are impacted by varying several parameters. Recall that the channel is populated with one or more DIMMs. Each DIMM supports one or more ranks—a rank is a collection of DRAM chips that work in unison to service a single command issued by the memory controller. For example, a rank may be composed of 16 DRAM chips, where each chip deals with a 4-bit input or output on every channel clock edge. Each such chip is referred to as a $\times 4$ (by four) chip. In other examples, a rank may be composed of 8×8 chips or 4×16 chips—note that in each case, a rank can handle data that are being placed on a 64-bit memory channel. A rank is itself partitioned into 8 (DDR3) or 16 (DDR4) banks. Each bank has a row buffer that essentially remembers the last row read out of a bank. Here's an example of a typical sequence of memory commands when performing a read from a bank: