parallelism took this a step further by providing more parallelism and hence more latency-hiding opportunities. It is likely that the use of instruction- and thread-level parallelism will be the primary tool to combat 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 memories, which we will see are used in GPUs. Such ideas have never made the mainstream 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.

Although one should be cautious about predicting the future of computing technology, history has shown that caching is a powerful and highly extensible idea that is likely to allow us to continue to build faster computers and ensure that the memory hierarchy can deliver the instructions and data needed to keep such systems working well.

2.9 Historical Perspective and References

In Section L.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, Naveen Muralimanohar, and Sheng Li

Case Study 1: Optimizing Cache Performance via Advanced Techniques

Concepts illustrated by this case study

- Non-blocking 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 is illustrated below:

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

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 two processor cycles. Assume that each iteration of the inner loop above requires four 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] <2.2> For the simple implementation given above, 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] <2.2> What should be the minimum size of the cache to take advantage of blocked execution?

b. [15] <2.2> How do the relative number of misses in the blocked and unblocked versions compare in the minimum sized cache above?

c. [15] <2.2> Write code to perform a transpose with a block size parameter B which uses $B \times B$ blocks.

d. [12] <2.2> What is the minimum associativity required of the L1 cache for consistent performance independent of both arrays' position in memory?

e. [20] <2.2> 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] <2.2> Assume you are designing a hardware prefetcher for the *unblocked* matrix transposition code above. The simplest type of hardware prefetcher only prefetches sequential cache blocks after a miss. More complicated "non-unit stride" hardware prefetchers can analyze a miss reference stream and detect and prefetch non-unit strides. In contrast, software prefetching can determine non-unit 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 non-unit stride prefetcher, in the steady state of the inner loop how many prefetches must be outstanding at a given time?

2.3 [15/20] <2.2> 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] <2.2> Create a blocked version of the matrix transpose with software prefetching.

b. [20] <2.2> 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

■ Crosscutting Issues: The Design of Memory Hierarchies

The program in Figure 2.29 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 2.29 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 2.29 was derived from a program written by Andrea Dusseau at the University of California–Berkeley

```
#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 2.29 C program for evaluating memory system.

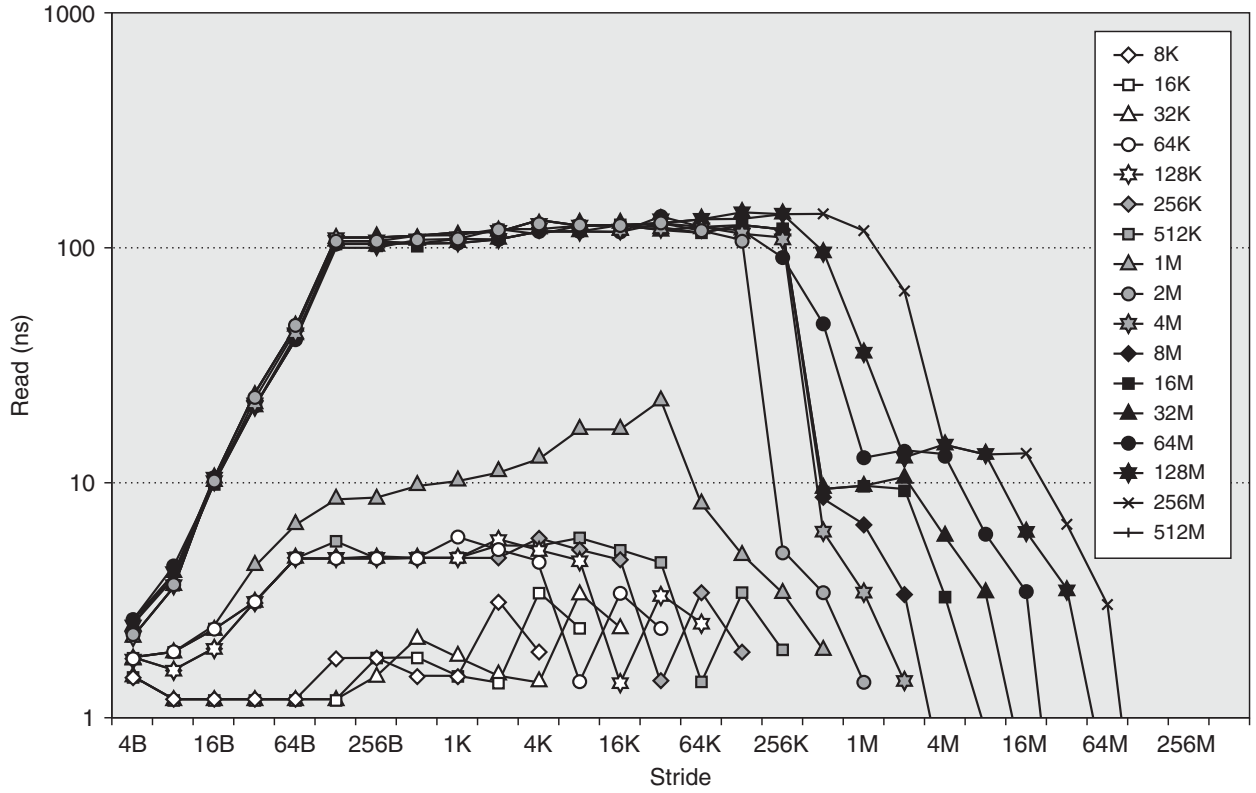

Figure 2.30 Sample results from program in Figure 2.29.

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

The program above 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 questions below, 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 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 2.30; 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 2.30:

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?

2.5 [12/15/15/20] <2.6> If necessary, modify the code in Figure 2.29 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 *translation lookaside buffer (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 2.29, 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 non obvious instructions from a piece of code. Try to use simple arithmetic instructions of known length in your instruction set architecture (ISA).

Exercises

2.8 [12/12/15] <2.2> The following questions investigate the impact of small and simple caches using CACTI and assume a 65 nm (0.065 µm) technology. (CACTI is available in an online form at *http://quid.hpl.hp.com:9081/cacti/*.)

a. [12] <2.2> Compare the access times of 64 KB caches with 64 byte blocks and a single bank. What are the relative access times of two-way and four-way set associative caches in comparison to a direct mapped organization?

b. [12] <2.2> Compare the access times of four-way set associative caches with 64 byte blocks and a single bank. What are the relative access times of 32 KB and 64 KB caches in comparison to a 16 KB cache?

c. [15] <2.2> For a 64 KB cache, find the cache associativity between 1 and 8 with the lowest average memory access time given that misses per instruction for a certain workload suite is 0.00664 for direct mapped, 0.00366 for two-way set associative, 0.000987 for four-way set associative, and 0.000266 for

eight-way set associative cache. Overall, there are 0.3 data references per instruction. Assume cache misses take 10 ns in all models. To calculate the hit time in cycles, assume the cycle time output using CACTI, which corresponds to the maximum frequency a cache can operate without any bubbles in the pipeline.

2.9 [12/15/15/10] <2.2> You are investigating the possible benefits of a way-predicting L1 cache. Assume that a 64 KB four-way set associative single-banked L1 data cache is the cycle time limiter in a system. As an alternative cache organization you are considering a way-predicted cache modeled as a 64 KB direct-mapped cache with 80% prediction accuracy. Unless stated otherwise, assume that a mispredicted way access that hits in the cache takes one more cycle. Assume the miss rates and the miss penalties in question 2.8 part (c).

a. [12] <2.2> What is the average memory access time of the current cache (in cycles) versus the way-predicted cache?

b. [15] <2.2> If all other components could operate with the faster way-predicted cache cycle time (including the main memory), what would be the impact on performance from using the way-predicted cache?

c. [15] <2.2> Way-predicted caches have usually been used only for instruction caches that feed an instruction queue or buffer. Imagine that you want to try out way prediction on a data cache. Assume that you have 80% prediction accuracy and that subsequent operations (e.g., data cache access of other instructions, dependent operations) are issued assuming a correct way prediction. Thus, a way misprediction necessitates a pipe flush and replay trap, which requires 15 cycles. Is the change in average memory access time per load instruction with data cache way prediction positive or negative, and how much is it?

d. [10] <2.2> As an alternative to way prediction, many large associative L2 caches serialize tag and data access, so that only the required dataset array needs to be activated. This saves power but increases the access time. Use CACTI's detailed Web interface for a 0.065 µm process 1 MB four-way set associative cache with 64 byte blocks, 144 bits read out, 1 bank, only 1 read/write port, 30 bit tags, and ITRS-HP technology with global wires. What is the ratio of the access times for serializing tag and data access in comparison to parallel access?

2.10 [10/12] <2.2> You have been asked to investigate the relative performance of a banked versus pipelined L1 data cache for a new microprocessor. Assume a 64 KB two-way set associative cache with 64 byte blocks. The pipelined cache would consist of three pipestages, similar in capacity to the Alpha 21264 data cache. A banked implementation would consist of two 32 KB two-way set associative banks. Use CACTI and assume a 65 nm (0.065 µm) technology to answer the following questions. The cycle time output in the Web version shows at what frequency a cache can operate without any bubbles in the pipeline.

a. [10] <2.2> What is the cycle time of the cache in comparison to its access time, and how many pipestages will the cache take up (to two decimal places)?

b. [12] <2.2> Compare the area and total dynamic read energy per access of the pipelined design versus the banked design. State which takes up less area and which requires more power, and explain why that might be.

2.11 [12/15] <2.2> Consider the usage of critical word first and early restart on L2 cache misses. Assume a 1 MB L2 cache with 64 byte blocks and a refill path that is 16 bytes wide. Assume that the L2 can be written with 16 bytes every 4 processor cycles, the time to receive the first 16 byte block from the memory controller is 120 cycles, each additional 16 byte block from main memory requires 16 cycles, and data can be bypassed directly into the read port of the L2 cache. Ignore any cycles to transfer the miss request to the L2 cache and the requested data to the L1 cache.

a. [12] <2.2> How many cycles would it take to service an L2 cache miss with and without critical word first and early restart?

b. [15] <2.2> Do you think critical word first and early restart would be more important for L1 caches or L2 caches, and what factors would contribute to their relative importance?

2.12 [12/12] <2.2> You are designing a write buffer between a write-through L1 cache and a write-back L2 cache. The L2 cache write data bus is 16 B wide and can perform a write to an independent cache address every 4 processor cycles.

a. [12] <2.2> How many bytes wide should each write buffer entry be?

b. [15] <2.2> What speedup could be expected in the steady state by using a merging write buffer instead of a nonmerging buffer when zeroing memory by the execution of 64-bit stores if all other instructions could be issued in parallel with the stores and the blocks are present in the L2 cache?

c. [15] <2.2> What would the effect of possible L1 misses be on the number of required write buffer entries for systems with blocking and nonblocking caches?

2.13 [10/10/10] <2.3> Consider a desktop system with a processor connected to a 2 GB DRAM with *error-correcting code (ECC)*. Assume that there is only one memory channel of width 72 bits to 64 bits for data and 8 bits for ECC.

a. [10] <2.3> How many DRAM chips are on the DIMM if 1 GB DRAM chips are used, and how many data I/Os must each DRAM have if only one DRAM connects to each DIMM data pin?

b. [10] <2.3> What burst length is required to support 32 B L2 cache blocks?

c. [10] <2.3> Calculate the peak bandwidth for DDR2-667 and DDR2-533 DIMMs for reads from an active page excluding the ECC overhead.

2.14 [10/10] <2.3> A sample DDR2 SDRAM timing diagram is shown in Figure 2.31. tRCD is the time required to activate a row in a bank, and column address strobe (CAS) latency (CL) is the number of cycles required to read out a column in a row Assume that the RAM is on a standard DDR2 DIMM with ECC, having 72 data lines. Also assume burst lengths of 8 which read out 8 bits, or a total of 64 B from

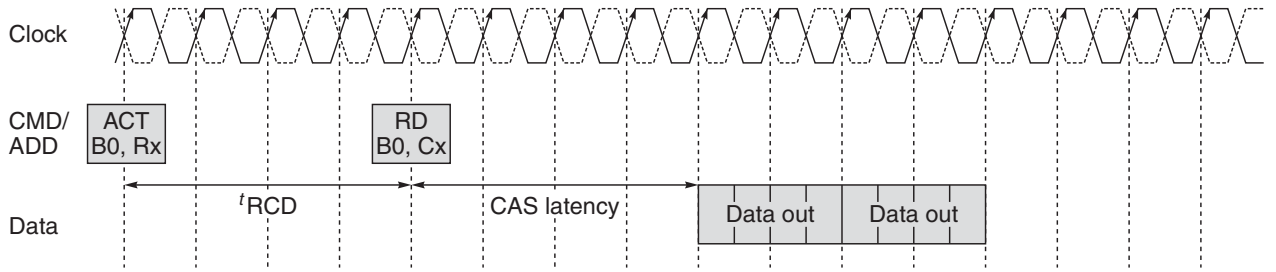

Figure 2.31 DDR2 SDRAM timing diagram.

the DIMM. Assume `tRCD = CAS (or CL) * clock_frequency`, and `clock_frequency = transfers_per_second/2`. The on-chip latency on a cache miss through levels 1 and 2 and back, not including the DRAM access, is 20 ns.

a. [10] <2.3> How much time is required from presentation of the activate command until the last requested bit of data from the DRAM transitions from valid to invalid for the DDR2-667 1 GB CL = 5 DIMM? Assume that for every request we automatically prefetch another adjacent cacheline in the same page.

b. [10] <2.3> What is the relative latency when using the DDR2-667 DIMM of a read requiring a bank activate versus one to an already open page, including the time required to process the miss inside the processor?

2.15 [15] <2.3> Assume that a DDR2-667 2 GB DIMM with CL = 5 is available for $130 and a DDR2-533 2 GB DIMM with CL = 4 is available for $100. Assume that two DIMMs are used in a system, and the rest of the system costs $800. Consider the performance of the system using the DDR2-667 and DDR2-533 DIMMs on a workload with 3.33 L2 misses per 1K instructions, and assume that 80% of all DRAM reads require an activate. What is the cost-performance of the entire system when using the different DIMMs, assuming only one L2 miss is outstanding at a time and an in-order core with a CPI of 1.5 not including L2 cache miss memory access time?

2.16 [12] <2.3> You are provisioning a server with eight-core 3 GHz CMP, which can execute a workload with an overall CPI of 2.0 (assuming that L2 cache miss refills are not delayed). The L2 cache line size is 32 bytes. Assuming the system uses DDR2-667 DIMMs, how many independent memory channels should be provided so the system is not limited by memory bandwidth if the bandwidth required is sometimes twice the average? The workloads incur, on an average, 6.67 L2 misses per 1K instructions.

2.17 [12/12] <2.3> A large amount (more than a third) of DRAM power can be due to page activation (see *http://download.micron.com/pdf/technotes/ddr2/TN4704.pdf* and *www.micron.com/systemcalc*). Assume you are building a system with 2 GB of memory using either 8-bank 2 GB x8 DDR2 DRAMs or 8-bank 1 GB x8 DRAMs, both with the same speed grade. Both use a page size of 1 KB, and the

last level cacheline size is 64 bytes. Assume that DRAMs that are not active are in precharged standby and dissipate negligible power. Assume that the time to transition from standby to active is not significant.

a. [12] <2.3> Which type of DRAM would be expected to provide the higher system performance? Explain why.

b. [12] <2.3> How does a 2 GB DIMM made of 1 GB x8 DDR2 DRAMs compare against a DIMM with similar capacity made of 1 Gb x4 DDR2 DRAMs in terms of power?

2.18 [20/15/12] <2.3> To access data from a typical DRAM, we first have to activate the appropriate row. Assume that this brings an entire page of size 8 KB to the row buffer. Then we select a particular column from the row buffer. If subsequent accesses to DRAM are to the same page, then we can skip the activation step; otherwise, we have to close the current page and precharge the bitlines for the next activation. Another popular DRAM policy is to proactively close a page and precharge bitlines as soon as an access is over. Assume that every read or write to DRAM is of size 64 bytes and DDR bus latency (Data out in Figure 2.30) for sending 512 bits is Tddr.

a. [20] <2.3> Assuming DDR2-667, if it takes five cycles to precharge, five cycles to activate, and four cycles to read a column, for what value of the row buffer hit rate (r) will you choose one policy over another to get the best access time? Assume that every access to DRAM is separated by enough time to finish a random new access.

b. [15] <2.3> If 10% of the total accesses to DRAM happen back to back or contiguously without any time gap, how will your decision change?

c. [12] <2.3> Calculate the difference in average DRAM energy per access between the two policies using the row buffer hit rate calculated above. Assume that precharging requires 2 nJ and activation requires 4 nJ and that 100 pJ/bit are required to read or write from the row buffer.

2.19 [15] <2.3> Whenever a computer is idle, we can either put it in stand by (where DRAM is still active) or we can let it hibernate. Assume that, to hibernate, we have to copy just the contents of DRAM to a nonvolatile medium such as Flash. If reading or writing a cacheline of size 64 bytes to Flash requires 2.56 µJ and DRAM requires 0.5 nJ, and if idle power consumption for DRAM is 1.6 W (for 8 GB), how long should a system be idle to benefit from hibernating? Assume a main memory of size 8 GB.

2.20 [10/10/10/10/10] <2.4> Virtual Machines (VMs) have the potential for adding many beneficial capabilities to computer systems, such as improved total cost of ownership (TCO) or availability. Could VMs be used to provide the following capabilities? If so, how could they facilitate this?

a. [10] <2.4> Test applications in production environments using development machines?

b. [10] <2.4> Quick redeployment of applications in case of disaster or failure?

c. [10] <2.4> Higher performance in I/O-intensive applications?

d. [10] <2.4> Fault isolation between different applications, resulting in higher availability for services?

e. [10] <2.4> Performing software maintenance on systems while applications are running without significant interruption?

2.21 [10/10/12/12] <2.4> Virtual machines can lose performance from a number of events, such as the execution of privileged instructions, TLB misses, traps, and I/O. These events are usually handled in system code. Thus, one way of estimating the slowdown when running under a VM is the percentage of application execution time in system versus user mode. For example, an application spending 10% of its execution in system mode might slow down by 60% when running on a VM. Figure 2.32 lists the early performance of various system calls under native execution, pure virtualization, and paravirtualization for LMbench using Xen on an Itanium system with times measured in microseconds (courtesy of Matthew Chapman of the University of New South Wales).

a. [10] <2.4> What types of programs would be expected to have smaller slowdowns when running under VMs?

b. [10] <2.4> If slowdowns were linear as a function of system time, given the slowdown above, how much slower would a program spending 20% of its execution in system time be expected to run?

c. [12] <2.4> What is the median slowdown of the system calls in the table above under pure virtualization and paravirtualization?

d. [12] <2.4> Which functions in the table above have the largest slowdowns? What do you think the cause of this could be?

Benchmark	Native	Pure	Para
Null call	0.04	0.96	0.50
Null I/O	0.27	6.32	2.91
Stat	1.10	10.69	4.14
Open/close	1.99	20.43	7.71
Install sighandler	0.33	7.34	2.89
Handle signal	1.69	19.26	2.36
Fork	56.00	513.00	164.00
Exec	316.00	2084.00	578.00
Fork + exec sh	1451.00	7790.00	2360.00

Figure 2.32 Early performance of various system calls under native execution, pure virtualization, and paravirtualization.

2.22 [12] <2.4> Popek and Goldberg's definition of a virtual machine said that it would be indistinguishable from a real machine except for its performance. In this question, we will use that definition to find out if we have access to native execution on a processor or are running on a virtual machine. The Intel VT-x technology effectively provides a second set of privilege levels for the use of the virtual machine. What would a virtual machine running on top of another virtual machine have to do, assuming VT-x technology?

2.23 [20/25] <2.4> With the adoption of virtualization support on the x86 architecture, virtual machines are actively evolving and becoming mainstream. Compare and contrast the Intel VT-x and AMD's AMD-V virtualization technologies. (Information on AMD-V can be found at *http://sites.amd.com/us/business/it-solutions/virtualization/Pages/resources.aspx*.)

a. [20] <2.4> Which one could provide higher performance for memory-intensive applications with large memory footprints?

b. [25] <2.4> Information on AMD's IOMMU support for virtualized I/O can be found in *http://developer.amd.com/documentation/articles/pages/892006101.aspx*. What do Virtualization Technology and an input/output memory management unit (IOMMU) do to improve virtualized I/O performance?

2.24 [30] <2.2, 2.3> Since instruction-level parallelism can also be effectively exploited on in-order superscalar processors and *very long instruction word (VLIW)* processors with speculation, one important reason for building an out-of-order (OOO) superscalar processor is the ability to tolerate unpredictable memory latency caused by cache misses. Hence, you can think about hardware supporting OOO issue as being part of the memory system! Look at the floorplan of the Alpha 21264 in Figure 2.33 to find the relative area of the integer and floating-point issue queues and mappers versus the caches. The queues schedule instructions for issue, and the mappers rename register specifiers. Hence, these are necessary additions to support OOO issue. The 21264 only has L1 data and instruction caches on chip, and they are both 64 KB two-way set associative. Use an OOO superscalar simulator such as SimpleScalar (*www.cs.wisc.edu/~mscalar/simplescalar.html*) on memory-intensive benchmarks to find out how much performance is lost if the area of the issue queues and mappers is used for additional L1 data cache area in an in-order superscalar processor, instead of OOO issue in a model of the 21264. Make sure the other aspects of the machine are as similar as possible to make the comparison fair. Ignore any increase in access or cycle time from larger caches and effects of the larger data cache on the floorplan of the chip. (Note that this comparison will not be totally fair, as the code will not have been scheduled for the in-order processor by the compiler.)

2.25 [20/20/20] <2.6> The Intel performance analyzer VTune can be used to make many measurements of cache behavior. A free evaluation version of VTune on both Windows and Linux can be downloaded from *http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/*. The program (`aca_ch2_cs2.c`) used in Case Study 2 has been modified so that it can work with VTune out of the box on Microsoft Visual C++. The program can be downloaded from *www.hpl.hp.com/*

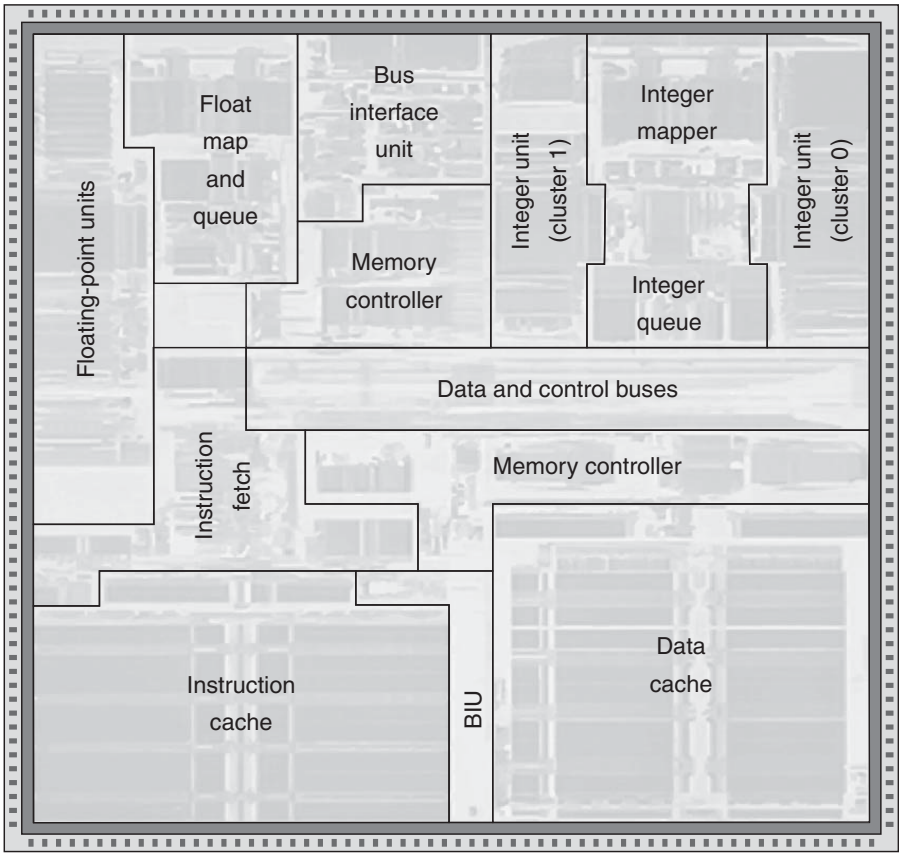

Figure 2.33 Floorplan of the Alpha 21264 [Kessler 1999].

research/cacti/aca_ch2_cs2_vtune.c. Special VTune functions have been inserted to exclude initialization and loop overhead during the performance analysis process. Detailed VTune setup directions are given in the README section in the program. The program keeps looping for 20 seconds for every configuration. In the following experiment you can find the effects of data size on cache and overall processor performance. Run the program in VTune on an Intel processor with the input dataset sizes of 8 KB, 128 KB, 4 MB, and 32 MB, and keep a stride of 64 bytes (stride one cache line on Intel i7 processors). Collect statistics on overall performance and L1 data cache, L2, and L3 cache performance.

a. [20] <2.6> List the number of misses per 1K instruction of L1 data cache, L2, and L3 for each dataset size and your processor model and speed. Based on the results, what can you say about the L1 data cache, L2, and L3 cache sizes on your processor? Explain your observations.

b. [20] <2.6> List the *instructions per clock* (IPC) for each dataset size and your processor model and speed. Based on the results, what can you say about the L1, L2, and L3 miss penalties on your processor? Explain your observations.

c. [20] <2.6> Run the program in VTune with input dataset size of 8 KB and 128 KB on an Intel OOO processor. List the number of L1 data cache and L2 cache misses per 1K instructions and the CPI for both configurations. What can you say about the effectiveness of memory latency hiding techniques in high-performance OOO processors? *Hint:* You need to find the L1 data cache miss latency for your processor. For recent Intel i7 processors, it is approximately 11 cycles.