6.823 Computer System Architecture

Problem Set #3

Spring *2002*

Students are strongly encouraged to collaborate in groups of up to three people. A group should hand in only one copy of the solution to the problem set. Problem sets are due in the beginning of class on the due date. To facilitate grading, the solution to each problem must be stapled separately. *Problem sets will not be accepted once solutions have been handed out.*

Problem 1: Optimizing for Cache Performance

In this problem, you will explore several common compiler optimizations for improving cache performance.

Problem 1.A Loop Ordering

Consider a 1 KB direct-mapped cache where each cache line contains four words. You are asked to consider the following two loops, written in C, which calculate the sum of the entries in a 64 by 16 matrix of 32-bit integers:

Loop A

```
for(i = 0; i < 64; i++)
  for(j = 0; j < 16; j++)
    sum += A[i][j];
```

Loop B

```
for(j = 0; j < 16; j++)
  for(i = 0; i < 64; i++)
    sum += A[i][j];
```

The matrix `A` is stored contiguously in memory, in row-major order[1], and is aligned to cache-line boundaries. You may assume that any other variables used are allocated to registers, and that the only cache activity involves the elements of the matrix.

Calculate the number of cache misses that occur when running loop A and when running loop B. Are the two values the same? Explain why or why not.

Problem 1.B Blocking

Blocking was discussed in lecture 8 as a way to optimize cache performance for computations on large matrices that do not fit in the cache. Here is a simple implementation of matrix multiply for an `N` by `N` matrix that does not use blocking:

```
for(i = 0; i < N; i++)
  for(j = 0; j < N; j++)
    {
      r = 0;

      for(k = 0; k < N; k++)
      {
        r = r + Y[i][k]*Z[k][j];
      }

      X[i][j] = r;
    }
```

where `r` is a register.

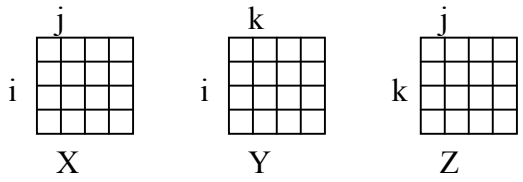
[1] Row-major order means that elements in the same row of the matrix are adjacent in memory.

Here is an implementation that uses blocking, with `B` × `B` blocks:

```
for(jj = 0; jj < N; jj += B)
  for(kk = 0; kk < N; kk += B)
    for(i = 0; i < N; i++)
      for(j = jj; j < min(jj+B, N); j++)
      {
        r = 0;

        for(k = kk; k < min(kk+B, N); k++)
        {
          r = r + Y[i][k]*Z[k][j];
        }

        X[i][j] += r;
      }
```



To see how blocking works, we will simulate the cache behavior for `N = 4` and `B = 2` for a fully-associative write-allocate cache with two-word cache lines using an LRU replacement policy. Assume that the matrices are aligned to cache line boundaries and that the compiler did not re-order any loads or stores. Complete Table 1-1 and Table 1-2 for the two implementations of matrix multiply, showing the progression of cache contents as accesses occur. Only fill in elements in the table when a value changes, or, when a cache hit occurs, in which case put "HIT" in the corresponding entry.

Calculate the miss rate for the two implementations based on the entries in the tables.

Problem 1.C More Blocking

Ben Bitdiddle is implementing the matrix-multiply-with-blocking algorithm from part B. He hypothesizes that he should be able to get reasonable performance if he can get the two inner-most loops of the algorithm to run without any cache misses other than compulsory misses.

Ben's implementation runs on the UltraSPARC-I, which has a direct-mapped, write-through non-allocating[2] cache. Ignoring conflict misses for the moment, how big does the data cache have to be if Ben were to use a blocking factor of `B`? Assume that the only cache activity involves the elements of the matrices (i.e. that any other variables used are allocated to registers).

[2] Another way of saying "no write allocate."

Using your result, calculate the maximum value of `B` that Ben could choose for his implementation. The UltraSPARC-I has a 16 KB data cache.

Ben's code operates on 3000 by 3000 matrices of 32-bit integer values, laid out in memory in row-major order. He runs some cache simulations and finds that for his particular data placement and cache configuration, there are some conflict misses, but not so many that it is a serious concern. However, when he tests the code on the UltraSPARC-I using the maximum block size you determined, the performance is dismal. Alyssa suggests that the problem may be related to the TLB.

Please explain how the TLB can be causing this problem. The TLB on the UltraSPARC-I is fully-associative and contains 64 entries. Pages are 8 KB. TLB misses are handled in software. You may assume that an LRU replacement policy is used. The data cache is virtually-tagged and virtually-indexed.

Problem 1.D Think Different

When Apple Computer first started using the PowerPC processor in their machines, they held a company-wide contest on who could write the fastest `BlockMove` routine, a library routine which copies some number of bytes from one memory location to another. [3]

The top two contestants were a microcoder and a computer architect. Their implementations of `BlockMove` were by far the fastest in the company. Ben Bitdiddle was interning at Apple and caught a glimpse of their code, and noticed that both routines use the `dcbz` PowerPC assembly instruction, which executes in one cycle, and has the following semantics:

```
dcbz rA,rB
```

If the block containing the byte addressable by (rA)+(rB) is in the data cache, all bytes of the block are cleared to zero.

If the block containing the byte addressed by (rA)+(rB) is not in the data cache, the block is allocated in the data cache without fetching the block from main memory, and all bytes of the block are set to zero.

Ben knows that all the 6.823 students have completed problem set 2 and are now experienced microcoders and architects. He wants to climb the ranks at Apple, and needs your help. Please explain how `dcbz` can be used in an optimized `BlockMove` routine. Another piece of information that may be important is that the PowerPC processor uses a write-allocate cache.

[3] `BlockMove` on the Macintosh is comparable to the C library function `memcpy`.

	1.C Table 1-1: Matrix Multiply Without Blocking									
	Line 0		Line 1		Line 2		Line 3		Line 4	
0	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
1	Y[0][0]	Y[0][1]								
2			Z[0][0]	Z[0][1]						
3		HIT								
4										
5										
6										
7										
8										
9										
10										
11										
12										
13										
14										
15										
16										
17										
18										

	1.C Table 1-2: Matrix Multiply With Blocking									
	Line 0		Line 1		Line 2		Line 3		Line 4	
0	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid	Invalid
1	Y[0][0]	Y[0][1]								
2			Z[0][0]	Z[0][1]						
3		HIT								
4										
5										
6										

7										
8										
9										
10										
11										
12										
13										
14										
15										
16										
17										
18										

Problem 2: Memory

Problem 2.A — Bank Assignment

Ben has a C program with an array definition of `unsigned int a[4][6]`. An `unsigned int` is 32 bits.

Assume that `a[0][0]` is at address 0x0 and that `a[0][1]` is at address 0x4. Show how the array elements map into the different memory banks. Each memory bank is 32-bits or 4 bytes wide.

bank number = (address/bank width in bytes) MOD NumBanks
address within bank = address MOD number of bytes in bank

Address within Bank	BANK 0	BANK 1	BANK 2	BANK 3	BANK 4
0x00	a[0][0]				
0x04					
0x08					
0x0C					
0x10					
0x14					
0x18					
0x1C					

Problem 2.B — Interleaved Memory

Assume the following description of a machine and cache performance

- 4 clocks to send address
- 24 clocks for the access time per word
- 4 clocks to send a word of data
- 1.2 memory accesses per instruction
- 1.5 average cycles per instruction (ignoring cache misses)
- cache block size of 1 word
- memory bus width of 1 word

What is the CPI for the given system if the miss rate is 3% for this configuration?

If we change the cache block size to two words, the miss rate falls to 2%. If we change the cache block size to four words, the miss rate falls to 1%. What is the CPI with two-word cache blocks and interleaved memory with two banks. What is the CPI with four-word cache blocks and four interleaved banks of memory?

What are the advantages and/or disadvantages of larger block sizes?

Problem 3: Virtual Memory Bits

In this problem we consider a DLX implementation with simple virtual memory enhancements. Each page table entry will map a virtual page number (VPN) to a physical page number (PPN). In addition to the page number translation, each page table entry also contains some permission/status bits.

Bit Name	Bit Definition
PPN / DBN	Physical Page Number / Disk Block Number
V (valid)	1 if the page table entry is valid, 0 otherwise
R (resident)	1 if the page is resident in memory, 0 otherwise
W (writable)	1 if the page is writable, 0 otherwise
U (used)	1 if the page has been accessed *recently*, 0 otherwise
M (modified)	1 if the page has been modified, 0 otherwise
S (supervisor)	1 if the page is only accessible in supervisor mode, 0 otherwise

Each entry in the TLB has a tag that is matched against the VPN and a TLB Entry Valid bit (note, the TLB Entry Valid bit is not the V bit shown in the table above). The TLB Entry Valid bit will be set if the TLB entry is valid. Each TLB entry also contains all the fields from the page table that are listed above.

A TLB miss (VPN does not match any of the tags for entries that have the TLB Entry Valid bit set) causes an exception. On a TLB miss kernel software will load the page table entry into the TLB and will restart the memory access. (Kernel software can modify anything in the TLB that it likes and always runs in supervisor mode). If the entry being replaced was valid, then the kernel will also write the TLB entry that is being replaced back to the page table.

Hardware will set the used bit whenever a TLB hit to the corresponding entry occurs. Similarly, the modified bit (in the TLB entry) will be set when a store to the page happens.

All exceptions that come from the TLB (hit or miss) are handled by software. For example, the possible exceptions are as follows:

TLB Miss:	VPN does not match any of tags for entries that have the TLB Entry Valid bit set.
Page Table Entry Invalid:	Trying to access a virtual page that has no mapping to a physical address.
Write Fault (Store only):	Trying to modify a read-only page (W is 0).
Protection Violation:	Trying to access a protected (supervisor) page while in user mode.
Page Fault:	Page is not resident.

(Unless noted, exceptions can occur for both loads and stores)

Problem 3.A

Whenever a TLB entry is replaced we write the entire entry back to the page table. Ben thinks this is a waste of memory bandwidth. He thinks only a few of the bits need to be written back. For each of the bits explain why or why not they need to be written back to the page table.

With this in mind, we will see how we can minimize the number of bits we actually need in each TLB entry throughout the rest of the problem.

Problem 3.B

Ben does not like the TLB design. He thinks the TLB Entry Valid bit should be dropped and the kernel software should be changed to ensure that all TLB entries are always valid. Is this a good idea? Explain the advantages and disadvantages of such a design.

Problem 3.C

Alyssa got wind of Ben's idea and suggests a different scheme to eliminate one of the valid bits. She thinks the page table entry valid and TLB Entry Valid bits can be combined into a single bit.

On a refill this combined valid bit will take the value that the page table entry valid bit had. A TLB entry is invalidated by writing it back to the page table and setting the combined valid bit in the TLB entry to invalid.

How does the kernel software need to change to make such a scheme work? How do the exceptions that the TLB produces change?

Problem 3.D

Now, Bud Jet jumps into the game. He wants to keep the TLB Entry Valid bit. However, there is no way he is going to have two valid bits in each TLB entry (one for the TLB entry one for the page table entry). Thus, he decides to drop the page table entry valid bit from the TLB entry.

How does the kernel software need to change to make this work well? How do the exceptions that the TLB produces change?

Problem 3.E

Compare your answers to Problem 3.C and 3.D. What scheme will lead to better performance?

Problem 3.F

How about the R bit? Can we remove them from the TLB entry without significantly impacting performance? Explain briefly.

Problem 3.G

The processor has a kernel (supervisor) mode bit. Whenever kernel software executes the bit is set. When user code executes the bit is not set. Parts of the user's virtual address space are only accessible to the kernel. The supervisor bit in the page table is used to protect this region—an exception is raised if the user tries to access a page that has the supervisor bit set.

Bud Jet is on a roll and he decides to eliminate the supervisor bit from each TLB entry. Explain how the kernel software needs to change so that we still have the protection mechanism and the kernel can still access these pages through the virtual memory system.

Problem 3.H

Alyssa P. Hacker thinks Ben and Bud are being a little picky about these bits, but has devised a scheme where the TLB entry does not need the M bit or the U bit. It works as follows. If a TLB miss occurs due to a load, then the page table entry is read from memory and placed in the TLB. However, in this case the W bit will always be set to 0. Provide the details of how the rest of the scheme works (what happens during a store, when do the entries need to be written back to memory, when are the U and M bits modified in the page table, etc.).

Problem 4: 64-bit Virtual Memory

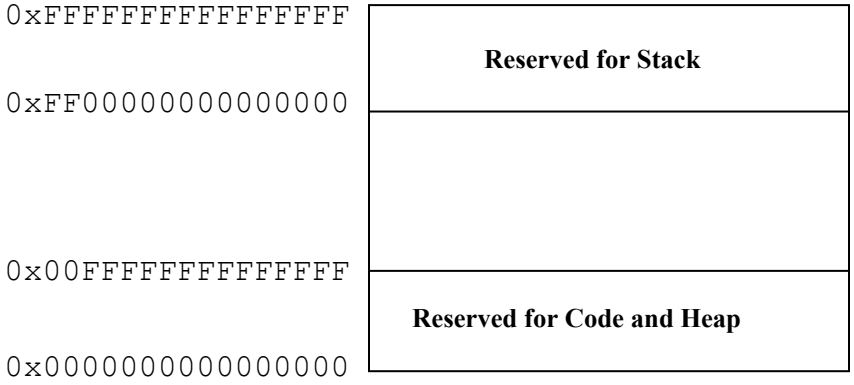
Ben is experimenting with the idea of using a 64-bit virtual address.

Problem 4.A

How large is the page table if only a single-level page table is used? Assume that each page is 4KB, each page table entry is 4 bytes, and that the processor is byte-addressable.

Problem 4.B

Ben read in a recent technical journal that many current implementations of 64-bit ISAs implement only part of the large virtual address space. They usually segment the virtual address space into three parts as shown below: one used for stack, one used for code and heap data, and the third one unused.



A special circuit is used to detect whether the top eight bits of an address are all zeros or all ones before the address is sent to the virtual memory system. If they are not all equal, an invalid virtual memory address trap is raised. This scheme in effect removes the top seven bits from the virtual memory address, but retains a memory layout that will be compatible with future designs that implement a larger virtual address space.

Ben likes the idea, but wants an even cheaper virtual memory system. He decided to remove the top 22 bits and only use the lower 42 bits to index the virtual memory. How large is the single-level page table now?

Problem 4.C

Ben is still unsatisfied about the page table size and asks you to use a three-level hierarchical page table that breaks the 42-bit address into three 10-bit page indices and a 12-bit page offset. If page table overhead is defined as (in bytes):

$$\frac{\text{PHYSICAL MEMORY USED BY PAGE TABLES FOR A USER PROCESS}}{\text{PHYSICAL MEMORY USED BY THE USER CODE, HEAP, AND STACK}}$$

What is the smallest possible page table overhead for the three-level hierarchical scheme? Remember that a complete page table page (1024 PTEs) is allocated even if only one PTE is used. Assume a large enough physical memory that no pages are ever swapped to disk.

What is the largest possible page table overhead for the three-level hierarchical scheme? Assume that once a user page is allocated in memory, the whole page is considered to be useful.

Problem 4.D

Alyssa P. Hacker is unhappy with the large hole in the virtual address space given by the proposed scheme. She decides that a hashed page table is the way to go. Again, the machine has a 64-bit virtual address and 4KB pages. The hardware paging system has only one page table with 64 slots, each containing 8 PTEs. Alyssa decides to use ***X mod 64*** as the hash function to select a slot, where ***X*** is the VPN. The page table resides in memory and Alyssa's design has no TLB, so each PTE read requires one memory access. During a page table lookup, all PTEs in each slot are searched sequentially. If there is a miss in the page table, a trap is raised and a software handler will refill the page table, with each refill requiring 10 memory accesses on average.

Alyssa runs a very simple benchmark that repeatedly loops over an array of 2^{21} bytes, reading one byte at a time in sequential address order. On average in the steady state, how many memory accesses are performed for each byte read by the user program? Ignore the memory traffic for instruction fetch, assume that the array starts on a page boundary, and there are no other memory accesses in the user code apart from the single byte memory accesses.

Problem 4.E

Alyssa now decides to add a one-entry TLB to the hashed paging system. What is the average number of memory accesses per user byte read for Alyssa's benchmark?