log2(nbytes)	log2(stride)	log2(nrefs)	time
10	6	30	2.755 seconds
11	6	30	2.576 seconds
12	6	30	2.451 seconds
13	6	30	2.500 seconds
14	6	30	2.408 seconds
15	6	30	3.566 seconds
16	6	30	18.093 seconds
17	6	30	18.048 seconds
18	6	30	18.150 seconds
19	6	30	18.931 seconds
20	6	30	39.302 seconds
21	6	30	67.068 seconds

Table 1: Run times for the cacheSize program

```
for(p = a; p < sizeGuess/sizeof(Thing); p++) /* initialize array */
  *p = (Thing)(p + stride); /* a pointer to the element stride away */
a[-stride] = (Thing)(a);
/* the loops */
p = a;
n_outer = 1 ≪ (l2nrefs - 4);
for(i = 0; i < n_outer; i++) {
  p = (Thing *)(*p); p = (Thing *)(*p);
  . . . /* repeat for a total of 16 accesses through *p */
}
printf("p = %d\n", p);
```

Figure 1: Pointer chasing version of the cacheSize program

CpSc 418: Homework 3

Solutions

1. Measuring cache parameters

   (a) L1 size and miss penalty:
   I tried running the cacheSize program on a 600MHz, Pentium III running Linux. Table 1 shows the runtimes that I observed. The execution time increased noticibly at 2^{16} bytes, suggesting a L1 cache size of $2^{15} = 64$K bytes.

   However, I was curious about the slight increase at 2^{15} bytes and suspected that there might be a problem such as the superscalar processor issueing multiple loads to hide the L2 cache miss latency. To prevent this possibility, I modified the cacheSize program so that each reference had to complete before then next one started. In particular, I defined a type called Thing so I could easily change the size of the elements of the array:

   ```
   typedef long long int Thing;
   ```

   and I adjusted the declarations, changed the cast on the malloc call. I rewrote the main loop as shown in figure 1 Table 2 shows the results of executing this version. Now, there's a big jump from the 2^{14} byte array to the 2^{15} byte one. This suggests a 16Kbyte L1 cache. The jump at 2^{20} bytes suggests that a L2-cache size of 512 Kbytes. The increase at 2^{18} bytes might be due to the TLB size. Or maybe it's the other way around.

log2(nbytes)	log2(stride)	log2(nrefs)	time
10	6	30	5.400 seconds
11	6	30	5.392 seconds
12	6	30	5.396 seconds
13	6	30	5.398 seconds
14	6	30	5.404 seconds
15	6	30	43.220 seconds
16	6	30	43.203 seconds
17	6	30	43.238 seconds
18	6	30	43.402 seconds
19	6	30	110.998 seconds
20	6	30	150.492 seconds
21	6	30	153.873 seconds
22	6	30	153.675 seconds
23	6	30	153.746 seconds

Table 2: Run times for the pointerChase program

log2(tlbGuess)	time
4	5.412
5	5.425
6	5.417
7	14.468
8	14.451
9	29.523

Table 3: Run times for TLB measurement program

I conclude that the L1 cache size is $2^14 = 16K$ bytes, and the L1 miss penalty is given by $(43.2 - 5.4)\text{sec}/(2^{30}\text{references})$, which is about 35 nano-seconds per reference.

I now started to question my earlier assumption that the anomolous results with cacheSize were due the superscalar supporting multiple misses. It would have to support rougly 43seconds/2.5seconds $\approx$ 17 outstanding references for this explanation to make sense. I then suspected some kind of prefetching and ran the original cacheSize program with a larger stride. Table 2 shows the result. These data confirm the 16K L1 cache size estimate. The apparent miss penalty drops to 11 nanoseconds. This suggests that the Pentium III can support three or four outstanding L1 misses (if there are no dependencies to prevent computing the future addresses).

Grading: This required more analysis than I had anticipated. Full credit will be given for any answer that observes a significant increase in execution size and uses that to derive a reasonable cache size (the deriviation must be sound..)

(b) L2 size and miss penalty:
The tables show a significant jump in execution time going from 2^{19} to 2^{20} bytes, and another increase when going from 2^{20} to 2^{21} bytes. This suggests an L2 cache size of either 512 Kbytes or 1 Mbyte. The other jump probably reflects TLB misses.

To separate the effects of TLB misses and L2 misses, I wrote a program that cyclically accessed tlbGuess locations with a stride of blockSize + pageSize. I used my blockSize = 16 bytes result below, and used the system call getpagesize() to determine the page size (4096 bytes). Table 3 shows the run-times that I measured. This suggests a TLB size of $2^6 = 64$ entries. The jump at 2^9 is because TLB entries are cached in the L1 cache when fetched from main memory. Because each location accessed in this program needs a separate TLB entry, each location consumes *two* L1 cache blocks: one for the data, and one for the TLB

blockGuess	time
8	43.142
16	43.136
32	7.146
64	7.091
128	6.259

Table 4: Run times for the cacheBlock program

entry. I believe that the increase at 2^9 is due to extra conflict misses in the L1 cache.

An array of 2^{19} bytes requires $2^{19}/4096 = 128$ TLB entries. This confirms that the increase at in table 2 is due to TLB misses, and the L2 cache size is 2^19 bytes. The apparent L2 miss penalty is $(153 - 43)\text{sec}/(2^{30}\text{references})$. This is about 102 nano-seconds.

Conclusions. The L2 cache size is 512 Kbytes. The L2 miss penalty is roughly 100 nano-seconds. As a byproduct, the TLB size appears to be 64 entries, and the TLB miss penalty appears to be about 8.4 nano-seconds.

Grading: Give full credit for a reasonable L2 estimate that is supported by the data. Extra credit may be given for identifying TLB effects.

(c) L1 Block size:
For the next part, I determined that the cache is 4-way set associative. To measure the block size, I'll reference 4 blocks offset by the cache size, and then 4 more offset by the cache size plus blockGuess bytes. I allocate the array to make sure that base of the array is at a multiple of 2^12 (certainly larger than the block size). This makes sure that the first set of four references fill the first sizeof(Thing) bytes of the cache block. If blockGuess is less than the block size, each access in the second set will conflict with the corresponding access in the first set, and every access will be a miss. If blockGuess is greater than or equal to the block size, then every reference will be a hit.

My program is basically the same as that for the associativity test – the main differnce is in the code that sets up the pointers. Figure 2 shows the code, and table 4 shows the run-times that I observed when running the code. All runs set cacheSize to 16K, assoc to 4, and nref to 2^{30}. This indicates that the block size is greater than 8 bytes and less than 24 bytes. It's reasonable to assume that the block size is a power of 2. I conclude that the cache size iskkk

(d) L1 Associativity:
Here, my strategy is to repeatedly access assocGuess values that are separated by *cache-size* bytes. If assocGuess is less than or equal to the associativity, then all of the references will be cache hits. If assocGuess is greater than the associativity, they will all be misses.

However, if there is a victim cache, the method described above will measure the associativity of the victim cache rather than the L1. To solve this problem, my program accesses several sequences of assocGuess values before repeating. If the cache has adequate associativity, each of the sequences will fit in a different set. Otherwise, the total number of misses will be enough to overwhelm a victim cache.

Figures 3 and 4 shows my code, and table 5 shows run-times that I measured with this program. All runs were with s1 (the inner stride) set to 256 bytes, the cache size estimate set to 16Kbytes, and nref set to 2^{30}. For each run, n1 is $5 -$ assocGuess; in other words, 32 different addresses are accessed, likely to be larger than the capacity of any victim cache.

The data indicate that the L1 data cache is 4-way set-associative.

(e) Is there a victim cache?
This can be determined by re-running the cacheAssoc program with n1 = 1. I tried this. The run-times that I got were within 6 milliseconds of those reported for the associativity test in all cases; in other words, the effect was insignificant. I conclude that the Pentium III does not have a victim cache.

```
/* cacheBlock.c:
 *   A program for emperically determining the associativity and block
 *   sizeof a cache.
 *   Usage:
 *      cacheBlock log2(cacheSize) log2(assoc) log2(blockGuess) log2(nref)
 *
 *   This program allocates an array of Thing's of size (n1*s1 + n2*s2) bytes.
 *   It then executes a loop that reads addresses:
 *              A, A + cacheSize, ... A + (assoc-1)*cacheSize
 *              A + blockGuess, A + cacheSize + BlockGuess, ...
 *                      A + (assoc-1)*cacheSize + BlockGuess
 *   It executes this pattern enough times to get a total of nref references.
 */

#define ALIGN 4096

int main(int argc, char **argv) {
  ...
  /* Allocate the array */
  a = (Thing *)malloc(2*cacheSize*assoc + ALIGN);
  if(a == NULL) { ... }
  a = (Thing *)((((Thing)(a)) + (ALIGN-1)) & ~(ALIGN-1));

  /* set up the loops */
  gap = assoc*cacheSize + blockGuess;
  for(i = 0; i < 2; i++) {
    for(j = 0; j < assoc; j++) {
      p = a + (j*cacheSize + i*gap)/sizeof(Thing);
      if(j < assoc-1) q = p + cacheSize/sizeof(Thing);
      else if(i == 0) q = a + gap/sizeof(Thing);
      else q = a;
      *p = (Thing)(q);
    }
  }

  /* the loops */
  n_outer = nref/16;
  for(i3 = 0; i3 < n_outer; i3++) {
    p = (Thing*)(*p); p = (Thing*)(*p);
    ... /* repeat for a total of 16 references to *p */
  }
}
```

Figure 2: Program for measuring block size

n2	time
1	1.345
2	1.349
4	1.345
8	10.791
16	10.783

Table 5: Run times for the cacheAssoc program

```
/* cacheAssoc.c:
 *   A program for emperically determining the associativity and block
 *   sizeof a cache.
 *   Usage:
 *      cacheAssoc log2(n1) log2(s1) log2(assocGuess) log2(cacheSize) log2(nref)
 *
 *   This program allocates an array of Thing's of size (n1*s1 + assocGuess*cacheSize) bytes.
 *   It then executes a loop that reads addresses:
 *              A, A + s1, A + 2*s1, A + 3*s1, ..., A + (n1 - 1)*s1,
 *              A+cacheSize, A+cacheSize + s1, A+cacheSize + 2*s1, ..., A+cacheSize + (n1-1)*s1,
 *              ...
 *              A+(assocGuess-1)*cacheSize, A+(assocGuess-1)*cacheSize + s1, ... A+(assocGuess-1)*cacheSize + (n1-1)*s1
 *   It executes this pattern enough times to get a total of nref references.
 */

#include <stdlib.h>
#include <stdio.h>

typedef long long int Thing;

int main(int argc, char **argv) {
  register Thing *a,    /* the array */
        *atop,          /* the next integer pointer beyond the array */
        *p;             /* the current pointer into the array */

  register int i3,      /* loop index for the outer loop */
        n_outer;        /* how many times to execute the outer loop */

  long long int n1, assocGuess, nref, s1, cacheSize, i1, i2;
  Thing *q;

  /* get our parameters from the command line */
  ...

  /* Allocate the array */
  a = (Thing *)malloc(n1*s1 + assocGuess*cacheSize);
  if(a == NULL) {...}
```

Figure 3: Program for measuring cache associativity (part 1)

```
  /* set up the loops */
  atop = a + assocGuess*cacheSize/sizeof(Thing);

  for(i2 = 0; i2 < assocGuess; i2++) {
    for(i1 = 0; i1 < n1; i1++) {
      p = a + (i2*cacheSize + i1*s1)/sizeof(Thing);
      if(i1 < n1-1) q = p + s1/sizeof(Thing);
      else if(i2 < assocGuess-1) q = a + (i2+1)*cacheSize/sizeof(Thing);
      else q = a;
      *p = (Thing)(q);
    }
  }

  /* the loops */
  n_outer = nref/16;
  for(i3 = 0; i3 < n_outer; i3++) {
    p = (Thing*)(*p); p = (Thing*)(*p);
    p = (Thing*)(*p); p = (Thing*)(*p);
    p = (Thing*)(*p); p = (Thing*)(*p);
    p = (Thing*)(*p); p = (Thing*)(*p);
    p = (Thing*)(*p); p = (Thing*)(*p);
    p = (Thing*)(*p); p = (Thing*)(*p);
    p = (Thing*)(*p); p = (Thing*)(*p);
    p = (Thing*)(*p); p = (Thing*)(*p);
  }

  /* print p (so the compiler can't optimize it out of existence) */
  printf("p = %d\n", p);

  /* we made it */
  exit(0);
}
```

Figure 4: Program for measuring cache associativity (part 2)

2. Code optimization (30 points)

(a) Unoptimized Code.

I used gcc under RedHat Linux. Figure 5 shows the assembly code. I'll focus on the inner loop, since that's where most of the time is spent. The body for the loop is the block of code starting at label `L14`. The first three instructions at `L13` are the loop test. I figured this out by looking for the floating point operations `fmull` and `faddp` and working out from there.

This code shows very litte optimization. The compiler output alignment directives before branch targets, presumably to ensure that branch targets are cache-aligned. Other than that, there is no code motion, strength reduction, or induction variable eliminations as we discussed in class.

(b) Optimized Code.

Figure 6 shows the code when compiled with the `-O4` flag. The inner loop is the block of code starting at label 14. This shows most of the optimizations described in class:

Branch at bottom

The test is at the bottom of the loop and branches back to the entry point for the loop body. There is a separate test earlier to handle the case when the loop body is executed zero times.

Induction variable elimination

The `%eax` register holds the pointer into the `a[]` array. The `%ecx` register holds the index for the `b[]` array. The load of the element from `b[]` is done with the rather CISC'y instruction: `fldl (%ebx,%ecx,8)`. The x86 architecture uses a stack oriented processor for floating point operations. The `sum` variable is kept on the stack in the optimized code (it is pushed and popped each iteration of the unoptimized code).

The loop test is somewhat different that the way we did it in class. The value of `n` is loaded into `%edx` at the beginning of the loop, and `%edx` is decremented each iteration. While this seems wasteful, at first, the x86 requires an extra compare operation before the branch if the code were to use the "top" variable like we used for the MIPS in class.

Code motion

The calculation of `8*n` for the step size in the `b[]` array has been moved out of the loop and stored in register `%ecx`.

The `movl 12(%ebp), %ebx` instruction at the beginning of the loop is redundant as register `%ebx` is not modified by the loop body. The instruction could be moved out of the loop. I tried it. The routine still calculated matrix multiplication correctly, but it didn't run any faster. The compiler probably new what it was doing.

Strength reduction

Integer multiplications by the constant 8 (i.e. `sizeof(double)`) have been replaced by a shift instruction, `sall`, or by using load instructions that have special versions for power-of-two indices.

(c) Speed-up.

I ran the code with n = 100, calling the matrixMultiply function 100 times. Without optimization, it the code took 4.013 seconds. With optimization, it took 1.013 seconds. The speed up was a factor of 4. Alternatively, one can say that the optimized version was 300% faster.

```
.text
        .align 4
.globl matrixMultiply
        .type    matrixMultiply,@function
matrixMultiply:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ebx
        subl    $28, %esp
        nop
        movl    $0, -12(%ebp)           # for(i = 0; ...)
        .p2align 2
.L3:    movl    -12(%ebp), %eax         # for(...; i < n; ...)
        cmpl    20(%ebp), %eax
        jl      .L6                     # i < n
        jmp     .L4                     # i >= n
        .p2align 2
.L6:    movl    $0, -16(%ebp)           # for(j = 0; ...)
        .p2align 2
.L7:     movl    -16(%ebp), %eax
        cmpl    20(%ebp), %eax          # for(...; j < n; ...)
        jl      .L10
        leal    -12(%ebp), %eax         # j >= n
        incl    (%eax)                  # i++
        jmp     .L3
        .p2align 2
.L10:   movl    $0, -32(%ebp)
        movl    $0, -28(%ebp)
        movl    $0, -20(%ebp)
        .p2align 2
.L11:   movl    -20(%ebp), %eax         # %eax <- k
        cmpl    20(%ebp), %eax          # for(...; k < n; ...)
        jl      .L14
        movl    -12(%ebp), %eax         # k >= n, store sum in c[i*n + j]
        imull   20(%ebp), %eax          # this code calculate &(c[i*n+j])
        addl    -16(%ebp), %eax         # then loads sum, 4 bytes at a time
        imull   $8, %eax, %ebx          # and stores them into c[...]
        ...
        leal    -16(%ebp), %eax         # j++
        incl    (%eax)
        jmp     .L7
        .p2align 2
.L14:   movl    -12(%ebp), %eax         # %eax <- i
        imull   20(%ebp), %eax          # %eax *= n  (i*n)
        addl    -20(%ebp), %eax         # %eax += k  (i*n+k)
        imull   $8, %eax, %ebx          # %ebx *= 8  ((i*n+k)*sizeof(double))
        movl    8(%ebp), %ecx           # %ecx <- a
        movl    -20(%ebp), %eax         # %eax <- k
        imull   20(%ebp), %eax          # %eax *= n  (k*n)
        addl    -16(%ebp), %eax         # %eax += j  (k*n + j)
        imull   $8, %eax, %edx          # %edx *= 8  ((k*n+j)*sizeof(double))
        movl    12(%ebp), %eax          # %eax <- b
        fldl    (%ecx,%ebx)             # push st,  a[i*n + k]
        fmull   (%eax,%edx)             # top_of_stack *= b[k*n + j]
        fldl    -32(%ebp)               # push st, sum
        faddp   %st, %st(1)             # add st(0), st(1)
        fstpl   -32(%ebp)               # store sum
        leal    -20(%ebp), %eax         # %eax <- &k
        incl    (%eax)                  # (*k)++
        jmp     .L11
        .p2align 2
.L4:    movl    16(%ebp), %eax          # return(c)
        movl    -4(%ebp), %ebx
        leave
        ret
```

Figure 5: Unoptimized assembly code for matrixMultiply

```
.text
        .align 4
.globl matrixMultiply
        .type    matrixMultiply,@function
matrixMultiply:
        pushl   %ebp                    # save registers
        movl    %esp, %ebp
        pushl   %edi
        pushl   %esi
        pushl   %ebx
        subl    $16, %esp
        movl    20(%ebp), %edi          # %edi <- n
        movl    $0, -16(%ebp)           # for(i = 0; ...)
        cmpl    %edi, -16(%ebp)         # i < n ?
        jge     .L18                    # i >= n, exit
        fldz
        movl    $0, -24(%ebp)           # j = 0
        .p2align 2
.L6:
        xorl    %esi, %esi              # %esi <- 0
        cmpl    %edi, %esi              # 0 >= n ?
        jge     .L19                    # 0 > n, exit
        movl    16(%ebp), %edx          # %edx <- c
        movl    -24(%ebp), %eax         # %eax <- j
        leal    (%edx,%eax,8), %eax     # %eax <- &(c[j])
        movl    -24(%ebp), %edx         # %edx <- j
        sall    $3, %edx                # %edx <<= 3
        fld     %st(0)
        movl    %eax, -28(%ebp)
        movl    %edx, -20(%ebp)
        .p2align 2
.L10:
        testl   %edi, %edi
        fld     %st(0)
        jle     .L20
        movl    -20(%ebp), %eax
        addl    8(%ebp), %eax
        movl    %esi, %ecx
        movl    %edi, %edx
        .p2align 2
.L14:
        movl    12(%ebp), %ebx          # %ebx <- b
        fldl    (%ebx,%ecx,8)           # push st(0) <- bb[q]
        fmull   (%eax)                  # st(0) *- *aa
        addl    %edi, %ecx              # q += n
        addl    $8, %eax                # aa <- aa + 1
        decl    %edx                    # p--
        faddp   %st, %st(1)             # sum += *aa * bb[q]
        jne     .L14                    # while(p > 0)
.L20:
        movl    -28(%ebp), %eax
        ...
```

Figure 6: Optimized Assembly code for matrixMultiply

3. Superscalar execution

(a) R10000 with multiple loads per cycle.
Here's the code from the lecture notes:

```
L1:     fld   $fa, 0($aa)
        addiu $aa, $aa, 8
        fld   $fb, 0($bb)
        addiu $bb, $bb, $n8
        fmult $fx, $fa, $fb
        bne   $aa, $atop, L1
        fadd  $sum, $sum# $fx delay slot
```

If multiple loads could be executed per cycle, this code would still require two cylces per iteration. This is because the MIPS R10000 can only fetch 4 instruction per cycle, and only decode 4 instructions per cycle. No matter how the instructions are aligned on a cache line, two cycles are required to fetch the instructions of the loop body for an iteration. Thus, the coalescing of loads does not help for this code.

(b) Execution with eight instructions/cycle fetch and decode.
I'll assume that the data is held in the L1 cache – the analysis in class ignored cache misses. The L1 cache has a blocksize of 32 bytes. This holds 4 doubles. As described in class, the pointer calculations can run ahead of the rest of the code. So, it's reasonable to assume lots of pending addresses in the queues. The processor will be able to fetch four loads from the a[] array in a single cycle, but only one load from the b[] array. Thus, five cycles are needed for the loads for five loop iterations. The rest of the functional units can keep up with this. Thus, the average time per loop body iteration is $5/4$ cycles.

The speed up is by a factor of $2/(5/4) = 1.6$. Alternatively, you can call this a speed up of 60%.

4. Snooping caches
Call the set of elements updated by a thread a tile. Each tile has $(N/4) \times (N/4)$ elements. The N elements on the perimeter of the tile are shared with other processors. In particular, each processor will try to write these N each iteration, and will have to gain exclusive access. Furthermore, the thread for the neighbouring tile (or tiles in the case of the elements on the corners) will read these converting the access to shared access. This creates two coherence misses per element per iterations. The total number of coherence misses is:

$$
\begin{array}{rrl}
 & 16 & \{ \text{number of processors} \} \\
* & 2 & \{ \text{a miss to read, and a miss to write} \} \\
* & N & \{ \text{number of elements on the border} \} \\
* & M & \{ \text{number of iterations} \} \\
= & 32MN & \{ \text{total number of misses} \}
\end{array}
$$

A similar calculation shows that the total number of references is $2MN^2$. Thus, the miss rate is $8/N$. For example, if $N = 100$, then 8% of all references will be coherence misses. If $N = 1000$, then only 0.8% of references will miss.