C + x86 Assembly : Optimizing Program Performance

By Amanda Falke
September 2014
Abstractmachines at gmail

Questions which I try to answer from :
Bryant and O'Halloran's *"Computer Systems: A Programmer's Perspective"*
Answers in **bold**

5.16 ◆
Write a version of the inner product procedure described in Problem 5.15 that uses four-way loop unrolling.
For x86-64, our measurements of the unrolled version give a CPE of 2.00 for integer data but still 3.00 for both single and double precision.
A. Explain why any version of any inner product procedure cannot achieve a CPE less than 2.00.

This is because the inner product procedure requires at least two operations - or two cycles - per element. This is demonstrated by the fact that, inside of the loop, we have this operation:

sum	**=**	**sum**	**+**	**udata[i]**	*****	**vdata[i];**
write to register		***read from register***		***read from memory: 1 CPE***		***read from memory: 1 CPE***
				1 CPE		***+ 1 CPE = 2 CPE***

B. Explain why the performance for floating-point data did not improve with loop unrolling.

The <u>model/reference machine's</u> computer architecture in chapter 5 of this book does not optimize for floating point operations.

5.18 ◆
Write a version of the inner product procedure described in Problem 5.15 that uses four-way loop unrolling along with reassociation to enable greater parallelism. Our measurements for this function give a CPE of 2.00 with x86-64 and 2.25 with IA32 for all types of data.

original:

```
/* Accumulate in temporary */
2 void inner4(vec_ptr u, vec_ptr v,
data_t *dest)
3 {
4 long int i;
5 int length = vec_length(u);
6 data_t *udata = get_vec_start(u);
7 data_t *vdata = get_vec_start(v);
8 data_t sum = (data_t) 0;
9
10 for (i = 0; i < length; i++) {
11 sum = sum + udata[i] * vdata[i];
12 }
13 *dest = sum;
14 }
```

Answer:

```
2 void inner4(vec_ptr u, vec_ptr v, data_t *dest)
3 {
4 long int i;
5 int length = vec_length(u);
6 int length4 = length - 3;

7 data_t *udata = get_vec_start(u);
8 data_t *vdata = get_vec_start(v);
9 data_t sum = (data_t) 0;
10
11 for ( i = 0; i < length4; i +=4 ) {
12 /* instead of the below, use
13 two way parallelism: */
14 // sum = sum + udata[i] * vdata[i];
15 // sum = sum + udata[i+1] * vdata[i+1];
16 // sum = sum + udata[i+2] * vdata[i+2];
17 // sum = sum + udata[i+3] * vdata[i+3];

18 /* two way parallelism: */
19 sum = sum + udata[i] * vdata[i]; // even
20 sum = sum + udata[i+1] * vdata[i+1]; //odd
21 } // first for loop ends

22 /* finish remaining elements */
23 for ( i = 0; i < length; i ++ ) {
24 sum = sum + udata[i] * vdata[i];
25 } // second, "finishing" for loop ends
26
27 *dest = sum; // register read, memory write
```

	28 } // function ends

5.22 ◆
Suppose you are given the task of improving the performance of a program consisting of three parts. Part A requires 20% of the overall run time, part B requires 30%, and part C requires 50%. You determine that for $1000 you could either speed up part B by a factor of 3.0 or part C by a factor of 1.5.Which choice would maximize performance?

Let's calculate:
B is 30% of time. Speedup by a factor of 3.0: 30% becomes 10%

C is 50% of time. Speedup by a factor of 1.5: 50% becomes 33%

Clearly, choosing to optimize B is a better choice than choosing to optimize C.

---------- CHAPTER 6 ----------

6.26 ◆
The following table gives the parameters for a number of different caches. For each cache, fill in the missing fields in the table. Recall that m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, E is the associativity, S is the number of cache sets, t is the number of tag bits, s is the number of set index bits, and b is the number of block offset bits.

	m	C	B	E	S	t	s	b
1	32	1024	4	4	**64**	**24**	**6**	**2**
2	32	1024	4	256	**1**	**30**	**0**	**2**
3	32	1024	8	1	**128**	**22**	**7**	**3**
4	32	1024	8	128	**1**	**29**	**0**	**3**
5	32	1024	32	1	**32**	**22**	**5**	**5**
6	32	1024	32	4	**8**	**24**	**3**	**5**

6.28 ◆
This problem concerns the cache in Problem 6.13.

A. List all of the hex memory addresses that will hit in set 1.

ANSWER:
For problem#6.13, (S E B M) = (8 2 4 13)

M = address size = 13 bit address:

12	11	10	9	8	7	6	5	4	3	2	1	0

Hence:

t = 8:								s = 3			b = 2:	
t	**t**	**t**	**t**	**t**	**t**	**t**	**t**	**s**	**s**	**s**	**b**	**b**

TAG BITS: In set 1, tag bits must be 45 = 0x45 = 0100 0101:

t = 8:								s = 3			b = 2:	
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	s	s	s	b	b

SET INDEX BITS: In set 1, s bits must be 1 = 001 = 0x1:

t = 8:								s = 3			b = 2:	
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	b	b

b BITS: In set 1, block offset b bits may vary, for values 0, 1, 2, or 3:

t = 8:								s = 3			b = 2:	
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**0**
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**0**	**1**
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**0**
0	**1**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**1**	**1**	**1**

B. List all of the hex memory addresses that will hit in set 6.

ANSWER:
For problem#6.13, (S E B M) = (8 2 4 13)

M = address size = 13 bit address:

12	11	10	9	8	7	6	5	4	3	2	1	0

Hence:

t = 8: **s = 3** **b = 2:**

t	t	t	t	t	t	t	t	s	s	s	b	b

TAG BITS: In set 6, tag bits must be 91 = 0x91 = 1001 0001:

t = 8: **s = 3** **b = 2:**

1	0	0	1	0	0	0	1	s	s	s	b	b

SET INDEX BITS: In set 6, s bits must be 6 = 110 = 0x6:

t = 8: **s = 3** **b = 2:**

1	0	0	1	0	0	0	1	1	1	0	b	b

b BITS: In set 6, block offset b bits may vary, for values 0, 1, 2, or 3:

t = 8: **s = 3** **b = 2:**

1	0	0	1	0	0	0	1	1	1	0	0	0

1	0	0	1	0	0	0	1	1	1	0	0	1

1	0	0	1	0	0	0	1	1	1	0	1	0

1	0	0	1	0	0	0	1	1	1	0	1	1

6.30 ◆◆
Suppose we have a system with the following properties:
. The memory is byte addressable.
. Memory accesses are to 1-byte words (not to 4-byte words).
. Addresses are 12 bits wide.
. The cache is two-way set associative (E = 2), with a 4-byte block size (B = 4) and four sets (S = 4).

see figure on page 633 of book for hex values for addresses, etc

A. The following diagram shows the format of an address (one bit per box). Indicate (by labeling the diagram) the fields that would be used to determine the following:
CO The cache block offset
CI The cache set index
CT The cache tag

11	10	9	8	7	6	5	4	3	2	1	0

ANSWER: C = S * E * B
C = 4 * 2 * 4 = 32 ....

s = 2 b = 2 t = (12-(2+2)) = t=8

CT	**CT**	**CT**	**CT**	**CT**	**CT**	**CT**	**CT**	**CI**	**CI**	**CO**	**CO**

B. For each of the following memory accesses indicate if it will be a cache hit or miss when carried out in sequence as listed. Also give the value of a read if it can be inferred from the information in the cache.

OPERATION	ADDRESS	HIT?	READ VALUE (OR UNKNOWN)
READ	0x834	**this is at address:** **1000 0011 0100** **set: 0x1 ...b offset: 0x0 ... tag: 0x83** **YES, IT IS A HIT**	**0xFE**
WRITE	0x836	**this is at address:** **1000 0011 0110** **set: 0x1 ... b offset: 0x2 ... tag: 0x83** **NOT A HIT. THIS IS A MISS**	**-**
READ	0xFFD	**this is at address:** **1111 1111 1101** **set: 0x3 ... b offset: 0x1 ... tag: 0xFF** **YES, THIS IS A HIT**	**0xC0**

6.31 ◆
Suppose we have a system with the following properties:
. The memory is byte addressable.
. Memory accesses are to 1-byte words (not to 4-byte words).
. Addresses are 13 bits wide.
. The cache is four-way set associative (E = 4), with a 4-byte block size (B = 4)
and eight sets (S = 8).
Consider the following cache state. SEE THE FIGURE ON PAGE 634 FOR TABLE
All addresses, tags, and values are given in hexadecimal format. The Index column contains the set index for each set of four lines. The Tag columns contain the tag value for each line. The V columns contain the valid bit for each line. The Bytes 0–3 columns contain the data for each line, numbered left-to-right starting with byte 0 on the left.

A. What is size (C) of this cache in bytes?

ANSWER: Since (S, E, B, M) = (8, 4, 4, 13) then s = 3, b = 2, m = 13, t = 8

So cache size is: C = S * E * B = 128

B. The box that follows shows the format of an address (one bit per box).
Indicate (by labeling the diagram) the fields that would be used to determine
the following:
CO The cache block offset CI The cache set index CT The cache tag

CT	**CT**	**CT**	**CT**	**CT**	**CT**	**CT**	**CT**	**CI**	**CI**	**CI**	**CO**	**CO**

6.32 ◆◆
Supppose that a program using the cache in Problem 6.31 references the 1-byte word at address 0x071A. Indicate the cache entry accessed and the cache byte value returned in hex. Indicate whether a cache miss occurs. If there is a cache miss, enter "–" for "Cache byte returned". Hint: Pay attention to those valid bits!

A. Address format (one bit per box):

ANSWER: NOTE THAT 0x071A = 0000 0111 0001 1010
And also, note that: s = 3, b = 2, m = 13, t = 8

t = 0011 1000 = 0x38 **s = 110 = 0x6** **b = 10 = 0x2**

0	**0**	**1**	**1**	**1**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**

B. Memory reference:

PARAMETER	VALUE
BLOCK OFFSET (CO)	**b = 10 = 0x2**
INDEX (CI)	**s = 110 = 0x6**
CACHE TAG (CT)	**t = 0011 1000 = 0x38**
CACHE HIT? Y / N	**yes, it's a hit!**
CACHE BYTE RETURNED	**0xF7**

6.35 ◆◆

Consider the following matrix transpose routine:
1 typedef int array[4][4];
2
3 void transpose2(array dst, array src)
4 {
5 int i, j;
6
7 for (i = 0; i < 4; i++) {
8 for (j = 0; j < 4; j++) {
9 dst[j][i] = src[i][j];
10 }
11
	Assume this code runs on a machine with the following properties:
	
	. sizeof(int) == 4.
	. The src array starts at address 0 and the dst array starts at address 64 (decimal).
	. There is a single L1 data cache that is direct-mapped, write-through, write-allocate, with a block size of 16 bytes.
	. The cache has a total size of 32 data bytes and the cache is initially empty.
	. Accesses to the src and dst arrays are the only sources of read and write misses, respectively.

A. For each row and col, indicate whether the access to src[row][col] and dst[row][col] is a hit (h) or a miss (m). For example, reading src[0][0] is a miss and writing dst[0][0] is also a miss.

DST ARRAY:

	COL 0	COL 1	COL 2	COL 3
ROW 0	miss	**hit**	**hit**	**hit**
ROW 1	**miss**	**hit**	**hit**	**hit**
ROW 2	**miss**	**hit**	**hit**	**hit**
ROW 3	**miss**	**hit**	**hit**	**hit**

SRC ARRAY:

	COL 0	COL 1	COL 2	COL 3
ROW 0	miss	**hit**	**hit**	**hit**
ROW 1	**miss**	**hit**	**hit**	**hit**
ROW 2	**miss**	**hit**	**hit**	**hit**
ROW 3	**miss**	**hit**	**hit**	**hit**

6.36 ◆◆

Repeat Problem 6.35 for a cache with a total size of 128 data bytes.

DST ARRAY:

	COL 0	COL 1	COL 2	COL 3
ROW 0	miss	**hit**	**hit**	**hit**
ROW 1	**miss**	**hit**	**hit**	**hit**
ROW 2	**miss**	**hit**	**hit**	**hit**
ROW 3	**miss**	**hit**	**hit**	**hit**

SRC ARRAY:

	COL 0	COL 1	COL 2	COL 3
ROW 0	miss	**hit**	**hit**	**hit**
ROW 1	**miss**	**hit**	**hit**	**hit**
ROW 2	**miss**	**hit**	**hit**	**hit**
ROW 3	**miss**	**hit**	**hit**	**hit**

6.37 ◆◆

This problem tests your ability to predict the cache behavior of C code. You are given the following code to analyze: 1 int x[2][128]; 2 int i; 3 int sum = 0; 4 5 for (i = 0; i < 128; i++) { 6 sum += x[0][i] * x[1][i]; 7 }	Assume we execute this under the following conditions: . sizeof(int) = 4. . Array x begins at memory address 0x0 and is stored in row-major order. . In each case below, the cache is initially empty. . The only memory accesses are to the entries of the array x. All other variables are stored in registers.

Given these assumptions, estimate the miss rates for the following cases:
A. Case 1: Assume the cache is 512 bytes, direct-mapped, with 16-byte cache blocks. What is the miss rate?
B. Case 2: What is the miss rate if we double the cache size to 1024 bytes?
C. Case 3: Now assume the cache is 512 bytes, two-way set associative using an LRU replacement policy, with 16-byte cache blocks.What is the cache miss rate?
D. For Case 3, will a larger cache size help to reduce the miss rate?Why or why not?
E. For Case 3, will a larger block size help to reduce the miss rate?Why or why not?

6.45 ◆◆◆
Download the mountain program from the CS:APP2Web site and run it on your favorite PC/Linux system. Use the results to estimate the sizes of the caches on your system.

ANSWER: I ran this on linux and received the following output. I'm not really sure about the output of this program.

machina@ada:~/fabcodez/dir1/a5$ **ls**
mountain mountain.tar
machina@ada:~/fabcodez/dir1/a5$ **cd mountain**
machina@ada:~/fabcodez/dir1/a5/mountain$ **ls**
clock.c clock.h fcyc2.c fcyc2.h Makefile mountain mountain.c README
machina@ada:~/fabcodez/dir1/a5/mountain$ **./mountain**
Clock frequency is approx. 2500.1 MHz
Memory mountain (MB/sec)
s1 s2 s3 s4 s5 s6 s7 s8 s9 s 10 s11 s12 s13 s14
s15 s16 s17 s18 s19 s 20 s21 s22 s23 s24 s25 s26 s27 s28
s29 s 30 s31 s32 s33 s34 s35 s36 s37 s38 s39 s 40
s41 s42 s43 s44 s45 s46 s47 s48 s49 s 50 s51 s52 s53 s54
s55 s56 s57 s58 s59 s 60 s61 s62 s63 s64
32m 6475.3 5697.9 4605.6 3695.9 2933.1 2397.9 2025.1 1767.1 1629.8 1 541.6
1474.4 1440.8 1460.5 1537.2 1667.5 1164.1 2049.2 2205.2 2453.7 2 591.2 2726.5
2843.9 2931.8 3053.4 3048.9 3073.3 3110.1 3114.6 3144.6 3 141.6 3138.0 936.2
3145.9 3143.8 3154.1 3158.2 3132.5 3146.1 3141.5 3 132.0 3136.2 3135.0 3213.2
3242.2 3271.2 3297.3 3308.6 3284.3 3354.3 3 377.4 3390.7 3405.1 3413.8
3423.9 3438.8 3445.5 3449.2 3445.7 3435.0 3 431.0 3430.6 3409.4 3393.8
950.4
16m 6497.2 6363.2 6078.9 5428.4 4696.9 4050.9 3529.8 3151.6 3145.1 3 6.5

..

I then tried to get the information programatically directly from the CPU and was denied, in these steps:

machina@ada:~/fabcodez/dir1/a5/mountain$ **/proc/cpuinfo**
-bash: /proc/cpuinfo**: Permission denied**
machina@ada:~/fabcodez/dir1/a5/mountain$

I then changed directories:

machina@ada:~/fabcodez/dir1/a5/mountain$ **cd /proc**

After I was granted permission, I did a list:
machina@ada:/proc$ **ls**
1 145 168 1944 2233 257 29002 32132 35956 5326 7741 buddyinfo
10 14587 1685 1945 2238 258 29005 32215 35974 54 7743 bus
101 14691 169 1946 2239 259 2901 32431 35976 5439 7744 cgroups
102 147 16972 195 224 26 29021 325 36 55 775 cmdline
103 1476 17 1956 225 260 291 32592 36228 56 776 consoles
1030 148 170 19659 2253 26040 29109 32661 36231 5646 7773 cpuinfo
10342 149 17090 1966 22610 262 292 32778 36987 57 78 crypto

In the case above, CPUINFO is listed in a TABLE of sorts, or a FILE TABLE of sorts, I believe. It appears that CPUINFO is entry number 1030.

To change directories to CPUINFO, simply cd to the directory 1030.

In my case, the CPUINFO directory was "table" number 101:
machina@ada:/proc$ **cd 101**
machina@ada:/proc/101$ **ls**
attr comm fd map_files net pagemap smaps task
autogroup coredump_filter fdinfo maps ns personality stack timers
auxv cpuset io mem numa_maps root stat wchan
cgroup cwd latency mountinfo oom_adj sched statm
clear_refs environ limits mounts oom_score schedstat status
cmdline exe loginuid mountstats oom_score_adj sessionid syscall
machina@ada:/proc/101$

In this way, I'm getting to know Linux and Unix, but still unable to really resolve this question.

machina@ada:/sys/firmware$ **ls**
acpi memmap
machina@ada:/sys/firmware$ **cd memmap**
machina@ada:/sys/firmware/memmap**$ ls**
0 1 10 11 12 2 3 4 5 6 7 8 9
machina@ada:/sys/firmware/memmap$ c**d ..**
machina@ada:/sys/firmware$ **cd** ..
machina@ada:/sys$ **ls**
block bus class dev devices firmware fs hypervisor kernel module power
machina@ada:/sys$ **cd kernel**
machina@ada:/sys/kernel$ ls
debug iommu_groups kexec_loaded profiling slab vmcoreinfo

fscache kexec_crash_loaded mm rcu_expedited uevent_helper
fscaps kexec_crash_size notes security uevent_seqnum
machina@ada:/sys/kernel$ **cd fscache**
machina@ada:/sys/kernel/fscache$ **ls**

Nothing available in fscache.

FINAL OBSERVATIONS:

Output of the mountain program problematic, as well as determining cache size using the mountain program.

When running the mountain program, output numbers differ according to the time of day I did the output.