Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Fall Term 2012

SYSTEMS PROGRAMMING AND COMPUTER ARCHITECTURE
Assignment 7: Performance Measurement and Optimization

Assigned on: **15th Nov 2012**
Due by: **22nd Nov 2012**

Part I [1]

Question 1: Associativity and CPE

Consider the following function for computing the product of an array of n integers. We have unrolled the loop by a factor of 3.

```
int aprod(int a[], int n)
{
  int i, x, y, z;
  int r = 1;
  for (i = 0; i < n-2; i += 3) {
    x = a[i]; y = a[i+1]; z = a[i+2];
    r = r * x * y * z;  // Product computation
  }
  for (; i < n; i++) {
    r *= a[i];
  }
  return r;
}
```

For the labeled `Product computation`, we can use parentheses to create five different associations of the computation, as follows:

```
r = ((r * x) * y) * z;  // A1
r = (r * (x * y)) * z;  // A2
r = r * ((x * y) * z);  // A3
r = r * (x * (y * z));  // A4
r = (r * x) * (y * z);  // A5
```

[1]Some of the questions of Part I are taken from the CS:APP textbook.

We measured the five versions of the function on an Intel Pentium III. The integer multiplication operation has a latency of 4 cycles and an issue time of 1 cycle on this machine.

The table that follows shows some values of the CPE and other values missing. The measured CPE values are those that were actually observed. “Theoretical CPE” means that performance that would be achieved if the only limiting factor were the latency and issue time of the integer multiplier.

Version	Measured CPE	Theoretical CPE
A1	4.00	
A2	2.67	
A3		
A4	1.67	
A5		

Fill in the missing entries. For missing values of the measured CPE, you can use the values from other versions that would have the same computational behavior. For the values of the theoretical CPE, you can determine the number of cycles that would be required for an iteration considering only latency and issue time of the multiplier, and then divide by 3.

Question 2: Conditional Move

A friend of yours has written an optimizing compiler that makes use of conditional move instructions. You try compiling the following C code:

```
int deref(int *xp)
{
  return xp ? *xp : 0;
}
```

The compiler generates the following code for the body of the procedure:

```
movl 8(%ebp), %edx    # Get xp
movl (%edx), %eax     # Get *xp as result
testl %edx, %edx      # Test xp
cmovzl %edx, %eax     # If 0, copy 0 to result
```

Is this a valid implementation of `deref`? Explain why (not).

Question 3: Optimized factorials

You've just joined a programming team that is trying to develop the world's fastest factorial routine. Starting with recursive factorial, they've converted the code to use iteration:

```
int fact(int n)
{
  int i, result = 1;

  for (i=n; i>0; i--)
    result = result * i;

  return result;
}
```

By doing so, they have reduced the number of CPE for the function from 63 to 4, measured on an Intel Pentium III. Still, they would like to do better.

Alice, the project manager, heard about loop unrolling, and told Dilbert to implement it. He generated the following code:

```
int fact_u2(int n)
{
  int i, result = 1;
  for (i=n; i>0; i-=2)
    result = (result*i) * (i-1);
  return result;
}
```

Unfortunately, the team discovered that this code returns 0 for some values of argument n.

a) What explains the big CPE difference between the recursive version and the iterative version?

b) For what values of n will $fact\_u2$ and $fact$ return different values?

c) Show how to fix $fact\_u2$.

d) Benchmarking $fact\_u2$ shows no improvement in performance. How would you explain that?

e) You modify the line inside the loop to read

```
result = result * (i * (i-1));
```

To everyone's astonishment, the measured performance now has a CPE of 2.5. How do you explain this performance improvement?

Question 4: Load-store interactions

Consider the following function to copy contents of one array to another:

```
void copy_array(int *src, int *dest, int n)
{
  int i;

  for (i = 0; i < n; i++) {
    dest[i] = src[i];
  }
}
```

Suppose `a` is an array of length 1000 initialized so that each element `a[i]` equals `i`.

a) What would be the effect of the call `copy_array(a+1,a,999)`?

b) What would be the effect of the call `copy_array(a,a+1,999)`?

c) Our performance measurements indicate that the first call has a CPE of 3.00, while the second call has a CPE of 5.00. To what factor do you attribute this performance difference?

d) What performance would you expect for the call `copy_array(a,a,999)`?

Part II

The provided sources describe a C program that calculates the matrix product for two pseudo-randomly generated $n \times n$ matrices for given n:

$$\begin{pmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{pmatrix} * \begin{pmatrix} b_{1,1} & \cdots & b_{1,n} \\ \vdots & & \vdots \\ b_{n,1} & \cdots & b_{n,n} \end{pmatrix} = \begin{pmatrix} c_{1,1} & \cdots & c_{1,n} \\ \vdots & & \vdots \\ c_{n,1} & \cdots & c_{n,n} \end{pmatrix}$$

with

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} \cdot b_{k,j}$$

Download the program from the course homepage and compile it. The syntax is `mmul` n, for instance,

```
./mmul 3
```

to compute the solution for two 3×3 matrices. Play around with the program.

Profiling

Use both the `time` command and `gprof` to measure the runtime for various values of n. Make a plot for each series of measurements. A link to an introduction to gprof can be found on the course website. **Hint:** In some shells, you need to type `/usr/bin/time` in order to start the program. Otherwise, a shell internal function will be used instead, which does not support some options, e.g. the `-f` flag.

Increasing Precision

You might encounter that the precision of `gprof` is not sufficient to measure the runtime of the separate functions for smaller values of n. Implement the `get_counter` function as described in the lecture.[2] Instrument your code in such a way that you get more precise information about the runtime of the individual functions.

Machine-Independent Optimization

Observe the code and detect the places where the code can be optimized. For each optimization that you apply to the code, make a precise measurement to estimate the influence of your optimization. The result of your optimization effort should be a plot showing the different steps and the impact on the runtime.

[2] See the slides of the very first lecture. You can also find material about using `rdtsc` in C on the web.

Reflection

Taking the insights so far, what is the main delimiting step in the program? Regarding the algorithmic complexity, which asymptotic runtime behavior would you expect? Compare your expected value with the experimental results.

Determining the Bottleneck

Consider the following two strategies to access a matrix:

- *Row major*: For each *row*, every *column* element is accessed in sequential order. Successively, the next row is processed.
- *Column major*: For each *column*, every *row* element is accessed in sequential order. Successively, the next column is processed.

Estimate the order of cache misses of the respective strategy. Have in mind how the matrices are stored in memory. What happens when n grows up to the size of a cache line? Check, if the program implements the better strategy. If this is not the case, change the program.

Advanced Optimization

A popular optimization technique for matrix multiplication is *blocking*. Instead of computing the whole matrix at once, the calculation is decomposed into the computation of smaller blocks. From the experience gathered in the last experiments, explain under which circumstances this leads to a performance improvement and mention which further optimization techniques can be applied.

Hand In Instructions

Hand in your paper solution (Part I) during your exercise class on the due date. Upload your source files and other digital material (Part II) to a subfolder **assignment7** of your SVN folder.