Performance Evaluation

May 2, 2000

Topics

• Getting accurate measurements
• Amdahl’s Law
“Time” on a Computer System

real (wall clock) time

= user time \( (\text{time executing instructing instructions in the user process}) \)

= system time \( (\text{time executing instructing instructions in kernel on behalf of user process}) \)

= some other user’s time \( (\text{time executing instructing instructions in different user’s process}) \)

+ + = real (wall clock) time

We will use the word “time” to refer to user time.
Time of Day Clock

- return elapsed time since some reference time (e.g., Jan 1, 1970)
- example: Unix gettimeofday() command
- coarse grained (e.g., ~3 μsec resolution on Linux, 10 msec resolution on Windows NT)
  - Lots of overhead making call to OS
  - Different underlying implementations give different resolutions

```c
#include <sys/time.h>
#include <unistd.h>

struct timeval tstart, tfinish;
double tsecs;
gettimeofday(&tstart, NULL);
P();
gettimeofday(&tfinish, NULL);
tsecs = (tfinish.tv_sec - tstart.tv_sec) +
       1e6 * (tfinish.tv_usec - tstart.tv_usec);
```
Interval (Count-Down) Timers

• set timer to some initial value
• timer counts down toward zero
• coarse grained (e.g., 10 msec resolution on Linux)

```c
void init_etime() {
    first.it_value.tv_sec = 86400;
    setitimer(ITIMER_VIRTUAL, &first, NULL);
}
```

```c
double get_etime() {
    struct itimerval curr;
    getitimer(ITIMER_VIRTUAL, &curr);
    return(double)(
        (first.it_value.tv_sec - curr.it_value.tv_sec) +
        (first.it_value.tv_usec - curr.it_value.tv_usec)*1e-6);
}
```

Using the interval timer

```c
init_etime();
secs = get_etime();
P();
secs = get_etime() - secs;
printf("%lf secs\n", secs);
```
Cycle Counters

• Most modern systems have built in registers that are incremented every clock cycle
  – Very fine grained
  – Maintained as part of process state
    » Save & restore with context switches
    » Counter will reflect time spent by user process

• Special assembly code instruction to access

• On (recent model) Intel machines:
  – 64 bit counter.
  – RDTSC instruction sets %edx to high order 32-bits, %eax to low order 32-bits

Wrap Around Times for 550 MHz machine

• Low order 32-bits wrap around every \(2^{32} / (550 \times 10^6) = 7.8\) seconds
• High order 64-bits wrap around every \(2^{64} / (550 \times 10^6) = 33539534679\) seconds
  – 1065.3 years
Using the Cycle Counter

• Example
  – Function that returns number of cycles elapsed since previous call to function
  – Express as **double** to avoid overflow problems

```c
/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

static double delta_cycles()
{
    unsigned ncyc_hi, ncyc_lo;
    double result;
    /* Get cycle counter as ncyc_hi and ncyc_lo */
    . . .
    /* Do double precision subtraction */
    . . .
    cyc_hi = ncyc_hi; cyc_lo = ncyc_lo;
    return result;
}
```
Accessing the Cycle Counter (cont.)

- GCC allows inline assembly code with mechanism for matching registers with program variables
- Code only works on x86 machine compiling with GCC

```c
unsigned nycy_h, nycy_l;
/* Get cycle counter */
asm("rdtsc\nmovl %edx, %0\nmovl %eax, %1"
    : "=r" (nycy_h), "=r" (nycy_l)
    : /* No input */
    : "%edx", "%eax");
```

- Emit assembly with `rdtsc` and two `movl` instructions
- Code generates two outputs:
  - Symbolic register `%0` should be used for `nycy_hi`
  - Symbolic register `%1` should be used for `nycy_lo`
- Code has no inputs
- Registers `%eax` and `%edx` will be overwritten
Accessing the Cycle Counter (cont.)

Emitted Assembly Code

delta_cycles:
    pushl %ebp        # Stack stuff
    movl %esp,%ebp
    pushl %esi
    pushl %ebx
    #APP
    rdtsc            # Result of ASM Statement
    movl %edx,%esi   # Uses %esi for ncy_c_hi
    movl %eax,%ecx  # Uses %ecx for ncy_c_lo
    #NO_APP
    movl %ecx,%ebx   # ncy_c_lo
    subl cyc_lo,%ebx
    cmpl %ecx,%ebx
    seta %al
    xorl %edx,%edx
    movb %al,%dl
    movl %esi,%eax   # ncy_c_hi
Using the Cycle Counter (cont.)

/* Keep track of most recent reading of cycle counter */
static unsigned cyc_hi = 0;
static unsigned cyc_lo = 0;

static double delta_cycles()
{
    unsigned ncyc_hi, ncyc_lo;
    unsigned hi, lo, borrow;
    double result;
    ...

    /* Do double precision subtraction */
    lo = ncyc_lo - cyc_lo;
    borrow = lo > ncyc_lo;
    hi = ncyc_hi - cyc_hi - borrow;
    result = (double) hi * (1 << 30) * 4 + lo;
    ...
}

Timing with Cycle Counter

double tsecs;
delta_cycles();
P();
tsecs = delta_cycles() / (MHZ * 1e6);
Measurement Pitfalls

Overhead

• Calling `delta_cycles()` incurs small amount of overhead
• Want to measure long enough code sequence to compensate

Unexpected Cache Effects

• artificial hits or misses
• e.g., these measurements were taken with the Alpha cycle counter:

  ```c
  foo1(array1, array2, array3);       /* 68,829 cycles */
  foo2(array1, array2, array3);       /* 23,337 cycles */
  vs.
  foo2(array1, array2, array3);       /* 70,513 cycles */
  foo1(array1, array2, array3);       /* 23,203 cycles */
  ```
Dealing with Overhead & Cache Effects

- Keep doubling number of times execute P() until reach some threshold
  - Used CMIN = 50000

```c
int cnt = 1;
double cmeas = 0;
double cycles;
do {
  int c = cnt;
P();          // Warm up cache */
(void) delta_cycles();
while (c-- > 0)
  P();
cmeas = delta_cycles();
cycles = cmeas / cnt;
cnt += cnt;
} while (cmeas < CMIN); /* Make sure have enough */
return cycles / (1e6 * MHZ);
```
Context Switching

Context switches can also affect cache performance

• e.g., (foo1, foo2) cycles on an unloaded timing server:
  » 71,002, 23,617
  » 67,968, 23,384
  » 68,840, 23,365
  » 68,571, 23,492
  » 69,911, 23,692

Why Do Context Switches Matter?

• Cycle counter only accumulates when running user process
• Some amount of overhead
• Caches polluted by OS and other user’s code & data
  – Cold misses as restart process

Measurement Strategy

• Try to measure uninterrupted code execution
Detecting Context Switches

Clock Interrupts

- Processor clock causes interrupt every $\Delta t$ seconds
  - Typically $\Delta t = 10$ ms
  - Same as interval timer resolution

Measurement takes place without interval timer advancing

- Can detect by seeing if interval timer has advanced during measurement

```c
start = get_et ime();

/* Perform Measurement */
...
if (get_et ime() - start > 0)
  /* Discard measurement */
```
Detecting Context Switches (Cont.)

External Interrupts

- E.g., due to completion of disk operation
- Occur at unpredictable times but generally take a long time to service

Detecting

- See if real time clock has advanced
  - Using coarse-grained interval timer

```c
start = get_rtime();

/* Perform Measurement */
...
if (get_rtime() - start > 0)
  /* Discard measurement */
```

Reliability

- Good, but not 100%
- Can’t get clean measurements on heavily loaded system
Improving Accuracy

Current Timer Code
- Assume that bad measurements always overestimate time
  - True if main problem is due to context switches
- Take multiple samples (2–10) until lowest two are within some small tolerance of each other

Better Timing Code
- Erroneous measurements both under- and over-estimate time, but are not correlated to each other
- Look for clustering of times among samples
Measurement Summary

It’s difficult to get accurate times
  • compensating for overhead
  • but can’t always measure short procedures in loops
    – global state
    – mallocs
    – changes cache behavior

It’s difficult to get repeatable times
  • cache effects due to ordering and context switches

Moral of the story:
  • Adopt a healthy skepticism about measurements!
  • Always subject measurements to sanity checks.
Amdahl’s Law

You plan to visit a friend in Normandy France and must decide whether it is worth it to take the Concorde SST ($3,100) or a 747 ($1,021) from NY to Paris, assuming it will take 4 hours Pgh to NY and 4 hours Paris to Normandy.

<table>
<thead>
<tr>
<th></th>
<th>time NY→Paris</th>
<th>total trip time</th>
<th>speedup over 747</th>
</tr>
</thead>
<tbody>
<tr>
<td>747</td>
<td>8.5 hours</td>
<td>16.5 hours</td>
<td>1</td>
</tr>
<tr>
<td>SST</td>
<td>3.75 hours</td>
<td>11.75 hours</td>
<td>1.4</td>
</tr>
</tbody>
</table>

Taking the SST (which is 2.2 times faster) speeds up the overall trip by only a factor of 1.4!
Speedup

Old program (unenhanced)

\[ T_1 \ + \ T_2 \]

Old time: \( T = T_1 + T_2 \)

New program (enhanced)

\[ T_1' = T_1 \ + \ T_2' \leq T_2 \]

New time: \( T' = T_1' + T_2' \)

Speedup: \( S_{\text{overall}} = \frac{T}{T'} \)

\( T_1 \) = time that can NOT be enhanced.

\( T_2 \) = time that can be enhanced.

\( T_2' \) = time after the enhancement.
Computing Speedup

Two key parameters:

\[
F_{\text{enhanced}} = \frac{T_2}{T} \quad \text{(fraction of original time that can be improved)}
\]
\[
S_{\text{enhanced}} = \frac{T_2}{T_2'} \quad \text{(speedup of enhanced part)}
\]

\[
T' = T_1' + T_2' = T_1 + T_2' = T(1-F_{\text{enhanced}}) + T_2'
\]
\[
= T(1 - F_{\text{enhanced}}) + \left(\frac{T_2}{S_{\text{enhanced}}}\right) \quad \text{[by def of } S_{\text{enhanced}}]\]
\[
= T(1 - F_{\text{enhanced}}) + T(F_{\text{enhanced}}/S_{\text{enhanced}}) \quad \text{[by def of } F_{\text{enhanced}}]\]
\[
= T((1 - F_{\text{enhanced}}) + F_{\text{enhanced}}/S_{\text{enhanced}})
\]

Amdahl’s Law:

\[
S_{\text{overall}} = \frac{T}{T'} = \frac{1}{(1 - F_{\text{enhanced}}) + F_{\text{enhanced}}/S_{\text{enhanced}}}
\]

Key idea:

- Amdahl’s Law quantifies the general notion of diminishing returns.
- It applies to any activity, not just computer programs.
Amdahl’s Law Example

Trip example:

- Suppose that for the New York to Paris leg, we now consider the possibility of taking a rocket ship (15 minutes) or a handy rip in the fabric of space-time (0 minutes):

<table>
<thead>
<tr>
<th></th>
<th>time NY-&gt;Paris</th>
<th>total trip time</th>
<th>speedup over 747</th>
</tr>
</thead>
<tbody>
<tr>
<td>747</td>
<td>8.5 hours</td>
<td>16.5 hours</td>
<td>1</td>
</tr>
<tr>
<td>SST</td>
<td>3.75 hours</td>
<td>11.75 hours</td>
<td>1.4</td>
</tr>
<tr>
<td>rocket</td>
<td>0.25 hours</td>
<td>8.25 hours</td>
<td>2.0</td>
</tr>
<tr>
<td>rip</td>
<td>0.0 hours</td>
<td>8 hours</td>
<td>2.1</td>
</tr>
</tbody>
</table>
Magnetic Disk Example

Disk surface spins at 3600*k RPM

Average Rotational Latency
- 1/2 revolution takes $1 / (120 \times k)$ seconds = $8.5/k$ milliseconds

Total Latency:
- $k = 1$: 14.5 ms 1.0X
- $k = 4$: 8.1 ms 1.8X
Lesson from Amdahl’s Law

Useful Corollary of Amdahl’s law:

\[ 1 \leq S_{overall} \leq \frac{1}{1 - F_{enhanced}} \]

<table>
<thead>
<tr>
<th>( F_{enhanced} )</th>
<th>Max ( S_{overall} )</th>
<th>( F_{enhanced} )</th>
<th>Max ( S_{overall} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1</td>
<td>0.9375</td>
<td>16</td>
</tr>
<tr>
<td>0.5</td>
<td>2</td>
<td>0.96875</td>
<td>32</td>
</tr>
<tr>
<td>0.75</td>
<td>4</td>
<td>0.984375</td>
<td>64</td>
</tr>
<tr>
<td>0.875</td>
<td>8</td>
<td>0.9921875</td>
<td>128</td>
</tr>
</tbody>
</table>

Moral: It is hard to speed up a program.

Moral++: It is easy to make premature optimizations.
Other Maxims

Second Corollary of Amdahl’s law:
• When you identify and eliminate one bottleneck in a system, something else will become the bottleneck

Beware of Optimizing on Small Benchmarks
• Easy to cut corners that lead to asymptotic inefficiencies
  – E.g., Intel’s string hash function