Performance Evaluation I
November 5, 1998

Topics
  • Performance measures (metrics)
  • Timing
  • Profiling
Performance expressed as a time

Absolute time measures (metrics)
- difference between start and finish of an operation
- synonyms: running time, elapsed time, response time, latency, completion time, execution time
- most straightforward performance measure

Relative (normalized) time measures
- running time normalized to some reference time
- (e.g. time/reference time)

Guiding principle: Choose performance measures that track running time.
Performance expressed as a rate

Rates are performance measures expressed in units of work per unit time.

Examples:

• millions of instructions / sec (MIPS)
• millions of fp instructions / sec (Mflops/sec) Mflop = 10^6 flops
• millions of bytes / sec (MBytes/sec) MByte = 2^20 bytes
• millions of bits / sec (Mbits/sec) Mbit = 10^6 bits
• images / sec
• samples / sec
• transactions / sec (TPS)
Performance expressed as a rate (cont)

Key idea: Report rates that track execution time.

Example: Suppose we are measuring a program that convolves a stream of images from a video camera.

Bad performance measure: MFLOPS

• number of floating point operations depends on the particular convolution algorithm: \( n^2 \) matrix-vector product vs \( n \log n \) fast Fourier transform. An FFT with a bad MFLOPS rate may run faster than a matrix-vector product with a good MFLOPS rate.

Good performance measure: images/sec

• a program that runs faster will convolve more images per second.
Timing mechanisms

Clocks
- returns elapsed time since epoch (e.g., Jan 1, 1970)
- Unix getclock() command
- coarse grained (e.g., $\mu s$ resolution on Alpha)

```c
long int secs, ns;
struct timespec *start, *stop;

getchek(DAY, start);
P();
getchek(DAY, stop);
secs = (stop->tv_sec - start->tv_sec);
ns = (stop->tv_nsec - start->tv_nsec);
printf("%ld ns\n", secs*1e9 + ns);
```
Timing mechanisms (cont)

Interval (count-down) timers

- set timer to some initial value
- timer counts down to zero, then sends Unix signal
- course grained (e.g., \textit{us} resolution on Alphas)

```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);
```
Timing mechanisms (cont)

Performance counters
• counts system events (CYCLES, IMISS, DMISS, BRANCHMP)
• very fine grained
• short time span (e.g., 9 seconds on 450 MHz Alpha)

```c
unsigned int counterRoutine[] = {
  0x601fc000u,
  0x401f0000u,
  0x6bfa8001u,
  0x6bfa8001u
};
unsigned int (*counter)(void) = (void *)counterRoutine;
```

Using the Alpha cycle counter
```c
cycles = counter();
P();
cycles = counter() - cycles;
printf("%d cycles\n", cycles);
```
Measurement pitfalls

Discretization errors
  • need to measure large enough chunks of work
  • but how large is large enough?

Unexpected cache effects
  • artificial hits or misses
  • cold start misses due to context swapping
The nature of time

real (wall clock) time

\[ \text{real (wall clock) time} = \text{user time (time executing instructing instructions in the user process)} \]

\[ \text{real (wall clock) time} = \text{system time (time executing instructing instructions in kernel on behalf of user process)} \]

\[ \text{real (wall clock) time} = \text{user time} + \text{system time} + \text{idle time} \]

*We will use the word “time” to refer to user time.*
Anatomy of a timer

**timer period**: \( dt \) secs/tick

**timer resolution**: \( 1/dt \) ticks/sec

Assume here that \( T_k = T_{k-1} + dt \)
Measurement pitfall #1: Discretization error

Actual program execution time:

- Measured time: \((T_n - T_1)\)
- Actual time: \((T_n - T_1) + (T_{\text{finish}} - T_n) - (T_{\text{start}} - T_1)\)
- Absolute error = measured time - actual time

\[
\begin{align*}
    f_{\text{start}} &= \frac{(T_{\text{start}} - T_1)}{dt} & \text{fraction of interval overreported} \\
    f_{\text{finish}} &= \frac{(T_{\text{finish}} - T_n)}{dt} & \text{fraction of interval underreported}
\end{align*}
\]

Absolute error = \(dt \ f_{\text{start}} - dt \ f_{\text{finish}} = dt \ (f_{\text{start}} - f_{\text{finish}})\)

Max absolute error = +/- \(dt\)
Examples of discretization error

Actual time = near zero
measured time = $dt$

Absolute measurement error = $+dt$
Examples of discretization error (cont)

Actual time = near $2dt$
measured time = $dt$

Absolute measurement error = $-dt$
Estimating the timer period $dt$

```c
start = 0;
while (start == (end = get_etime()))
    ;
dt = end - start;
printf("dt = %lf\n", dt);
```

Digital Unix Alpha systems: $dt = 1ms$
Modeling discretization error

Key idea: need to measure long enough to hide the discretization error.

Example:

```c
start = get_etime();
for (i=0; i<n; i++) {
    P();
}
tprime = get_etime() - start;
```

Question: how big must tprime be in order to get a “good” estimate of the running time of the loop?
Relative error analysis

Let $t$ and $t'$ be the actual and measured running times of the loop, respectively, and let $dt$ be the timer period.

Also, let $t'-t$ be the absolute error and let $|t'-t|/t$ be the relative error.

Problem: What value of $t'$ will result in a relative error less than or equal to $E_{\text{max}}$?

Fact (1): $|t'-t| \leq dt$
Fact (2): $t' - dt \leq t$

We want $|t'-t|/t \leq E_{\text{max}}$

\[
\frac{dt}{t} \leq E_{\text{max}} \quad (1)
\]
\[
\frac{dt}{E_{\text{max}}} \leq t \quad \text{(algebra)}
\]
\[
\frac{dt}{E_{\text{max}}} \leq t' - dt \quad (2)
\]

\[
\frac{dt}{E_{\text{max}}} + dt \leq t'
\]
Relative error analysis

```c
start = get_etime();
for (i=0; i<n; i++) {
    P();
}
tp = get_etime() - start; /* t' */
```

Example: Let $dt=1$ ms and $E_{max} = 0.05$ (i.e., 5% relative error)

Then:

$$\frac{dt}{E_{max}} + dt \leq t'$$

$$\frac{.001}{.05} + .05 \leq t'$$

$$t' \geq 0.070 \text{ seconds (70 ms)}$$
Measurement pitfall #2: Unexpected cache effects

Call ordering can introduce unexpected cold start misses (measured with Alpha cycle counter):

- \text{ip}(array1, array2, array3); /* 68,829 cycles */
- \text{ipp}(array1, array2, array3); /* 23,337 cycles */
- \text{ipp}(array1, array2, array3); /* 70,513 cycles */
- \text{ip}(array1, array2, array3); /* 23,203 cycles */

Context switches can alter cache miss rate

- 71,002 23,617 \text{(ip/ipp cycles on unloaded timing server)}
- 67,968 23,384
- 68,840 23,365
- 68,571 23,492
- 69,911 23,692
Measurement summary

It’s difficult to get accurate times
  • discretization error
  • 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.
Profiling

The goal of profiling is to account for the cycles used by a program or system.

Basic techniques

- **src translation**
  - gprof [Graham, 1982]

- **binary translation**
  - Atom [DEC, 1993]
  - pixie [MIPS, 1990]

- **direct simulation**
  - SimOS [Rosenblum, 1995]

- **statistical sampling**
  - prof (existing interrupt source)
  - SpeedShop [Zhaga, 1996] (performance counter interrupts)
# Profiling tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Overhead</th>
<th>Scope</th>
<th>Grain</th>
<th>Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>gprof</td>
<td>high</td>
<td>app</td>
<td>inst cnt</td>
<td>none</td>
</tr>
<tr>
<td>pixie</td>
<td>high</td>
<td>app</td>
<td>proc cnt</td>
<td>none</td>
</tr>
<tr>
<td>SimOS</td>
<td>high</td>
<td>sys</td>
<td>inst time</td>
<td>accurate</td>
</tr>
<tr>
<td>prof</td>
<td>low</td>
<td>app</td>
<td>inst time</td>
<td>none</td>
</tr>
<tr>
<td>DCPI</td>
<td>low</td>
<td>sys</td>
<td>inst time</td>
<td>accurate</td>
</tr>
<tr>
<td>SpeedShop</td>
<td>low</td>
<td>sys</td>
<td>inst time</td>
<td>inaccurate</td>
</tr>
</tbody>
</table>

**Overhead:** How much overhead (slowdown) does the tool introduce?  
**Scope:** Can the tool be used to profile an entire system, or a single program?  
**Grain:** What types of program units can the tool account for?  
**Stalls:** Can the tool account for instruction stalls?
Case study: DEC Continuous Profiling Infrastructure (DCPI)

Cutting edge profiling tool for Alpha 21164

Coarse grained profiling:

<table>
<thead>
<tr>
<th>cycles</th>
<th>%</th>
<th>procedure</th>
<th>image</th>
</tr>
</thead>
<tbody>
<tr>
<td>2064143</td>
<td>33.9</td>
<td>ffb8ZeroPolyArc</td>
<td>/usr/shlib/X11/lib_dec_ffb_ev5.so</td>
</tr>
<tr>
<td>517464</td>
<td>8.5</td>
<td>ReadRequestFromClient</td>
<td>/usr/shlib/X11/libos.so</td>
</tr>
<tr>
<td>305072</td>
<td>5.0</td>
<td>miCreateETandAET</td>
<td>/usr/shlib/X11/libmi.so</td>
</tr>
<tr>
<td>245450</td>
<td>4.0</td>
<td>bcopy</td>
<td>/vmunix</td>
</tr>
<tr>
<td>170723</td>
<td>2.8</td>
<td>in_checksum</td>
<td>/vmunix</td>
</tr>
</tbody>
</table>
DCPI case study

Fine-grained profiling:

<table>
<thead>
<tr>
<th>addr</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>pD</td>
<td>(p=branch mispredict)</td>
</tr>
<tr>
<td>pD</td>
<td>(D = data TLB miss)</td>
</tr>
<tr>
<td>009810</td>
<td>ldq t4,0(t1)</td>
</tr>
<tr>
<td>009814</td>
<td>addq t0,0x4,t0 (dual issue)</td>
</tr>
<tr>
<td>009818</td>
<td>ldq t5,8(t1)</td>
</tr>
<tr>
<td>00981c</td>
<td>ldq t6,16(t1)</td>
</tr>
<tr>
<td>009820</td>
<td>ldq a0,24(t1)</td>
</tr>
<tr>
<td>009824</td>
<td>lda t1, 32(t1) (dual issue)</td>
</tr>
<tr>
<td>dwD</td>
<td>(d=D-cache miss)</td>
</tr>
<tr>
<td>dwD</td>
<td>... 18</td>
</tr>
<tr>
<td>dwD</td>
<td>(w=write buffer)</td>
</tr>
<tr>
<td>009828</td>
<td>stq t4,0(t2)</td>
</tr>
<tr>
<td>00982c</td>
<td>cmpmult t0,v0,t4</td>
</tr>
<tr>
<td>009830</td>
<td>t5,8(t2)</td>
</tr>
<tr>
<td>s</td>
<td>(s=slotting hazard)</td>
</tr>
<tr>
<td>dwD</td>
<td></td>
</tr>
<tr>
<td>dwD</td>
<td>114.5</td>
</tr>
<tr>
<td>dwD</td>
<td></td>
</tr>
<tr>
<td>009834</td>
<td>stq t6,16(t2)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

for (i=0; i<n; i++)
c[i] = a[i];
DCPI architecture

- **16 bit cycle counter**
  - triggers interrupt on overflow
  - (user-writeable, distributed uniformly between 60K and 64 K for DCPI)

- **Sample histogram**
  - high priority performance counter interrupt handler

- **user level collection daemon**
  - (PC,PID,event)
DCPI architecture

Data Collection

• A performance counter counts occurrences of a particular kind of event (e.g., cycle counter counts the passage of instruction cycles).
• The performance counter triggers a high-priority hardware interrupt after a user-defined number of events (5200 times a second).
• The interrupt handler records a (PC, PID, event type) tuple.
• The interrupt handler counts samples with a hash table to reduce data volume.
• When hash table is full, handler passes results to user-level analysis daemon, which associates samples with load images.

Question:

• How to associate sample with load image?
  – modified /usr/sbin/loader records all dynamically loaded binaries.
  – modified exec call records all statically loaded binaries.
  – at startup, daemon scans for active processes and their regions.
DCPI architecture

Data Analysis

• evaluates PC sample histogram offline
• assigns cycles to procedures and instructions.
• identifies stalls and possible sources (data cache miss, write buffer overflow, TLB miss)
DCPI architecture

Interesting analysis problem:

If we see a large sample count for a particular instruction, did it

- execute many times with no stalls?
- execute only a few times with large stalls?

- **approach**: use compiler to identify basic blocks [a block of instructions with no jumps in (except possibly at the beginning) and no jumps out (except possibly at the end)]

- **compare** execution frequency of loads/stores to non-blocking instructions in the block.