# 15-213

# 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 matix-vector product vs nlogn 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., *u*s resolution on Alpha)

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

getclock(TIMEOFDAY, start);
P();
getclock(TIMEOFDAY, 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., *u*s resolution on Alphas)

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

```
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**

```
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)

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

**Using the Alpha cycle counter**

```
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

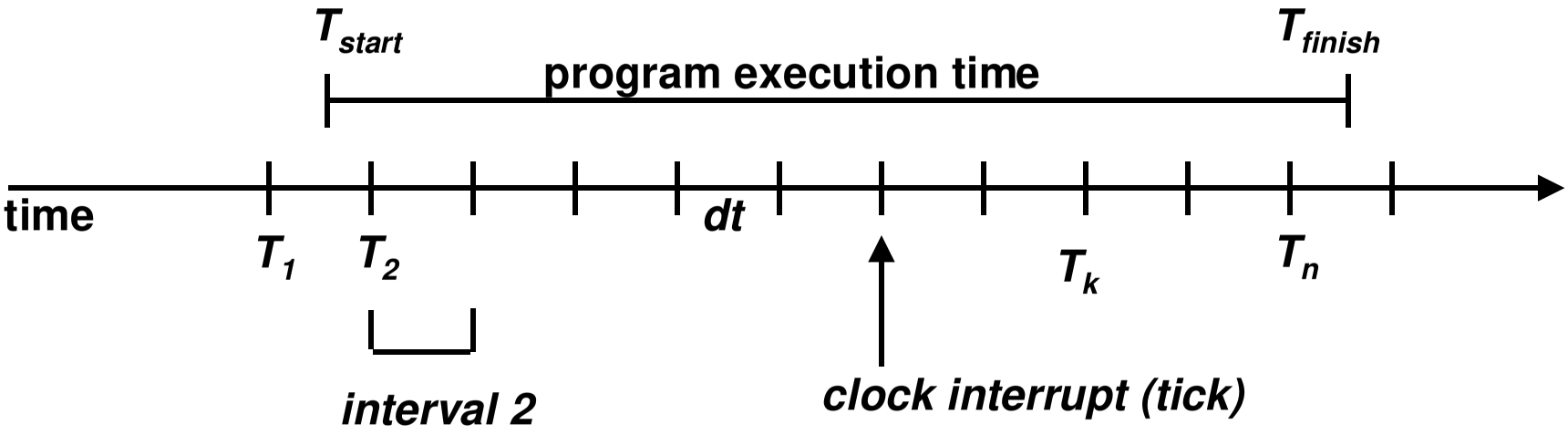
= user time (time executing instructing instructions in the user process)

= system time (time executing instructing instructions in kernel on behalf of user process

= real (wall clock) time

*We will use the word "time" to refer to user time.*

# Anatomy of a timer

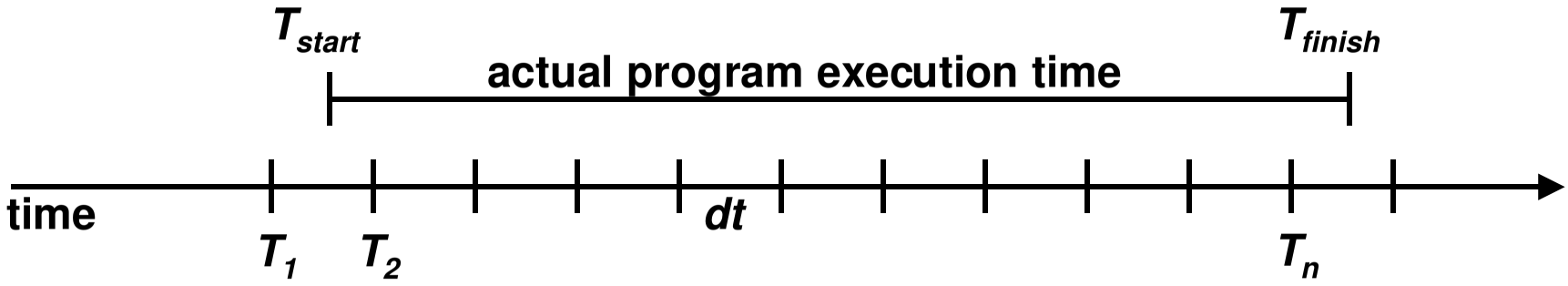



**timer period: *dt* secs/tick**

**timer resolution: *1/dt* ticks/sec**

**Assume here that $T_k = T_{k-1} + dt$**

# Measurement pitfall #1: Discretization error




**measured time:** $(T_n - T_1)$

**actual time:** $(T_n - T_1) + (T_{finish} - T_n) - (T_{start} - T_1)$

**absolute error = measured time - actual time**

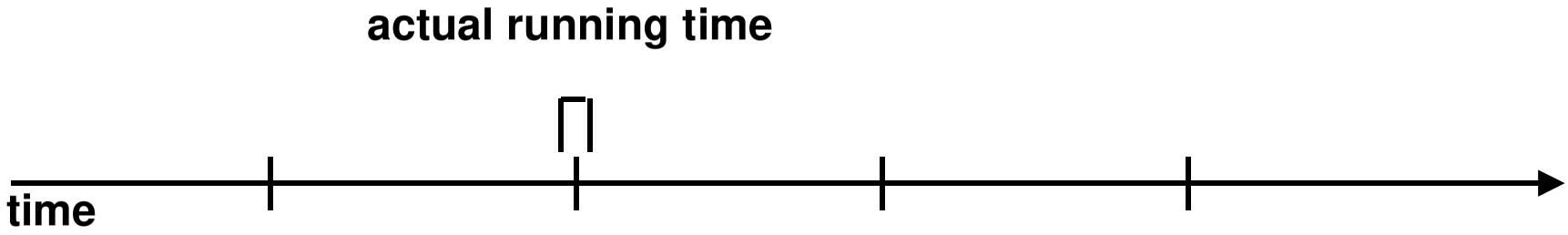
$f_{start} = (T_{start} - T_1)/dt$ **fraction of interval overreported**

$f_{finish} = (T_{finish} - T_n)/dt$ **fraction of interval underreported**

**absolute error =** $dt\ f_{start} - dt\ f_{finish} = dt\ (f_{start} - f_{finish})$

**max absolute error =** $+/- dt$

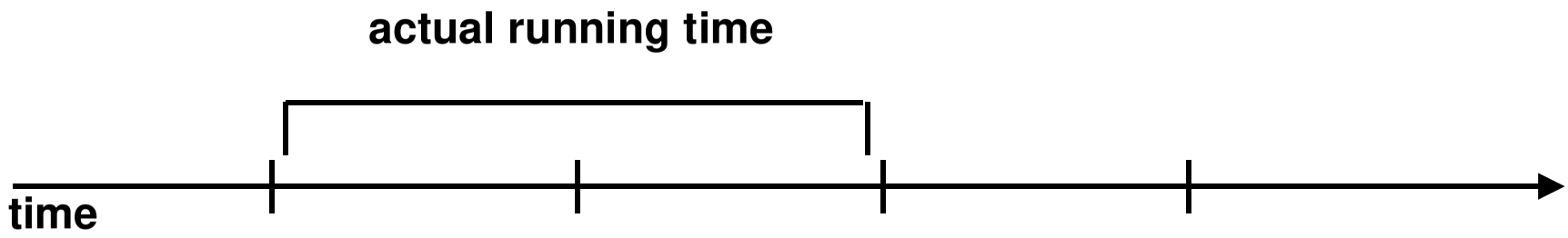
# Examples of discretization error




**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*

```
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:**

```
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 $E_{max}$?

Fact (1): */t'-t/ <= dt*
Fact (2): *t' - dt <= t*

We want */t'-t/t <= $E_{max}$*

| | |
|---|---|
| *dt/t <= $E_{max}$* | (1) |
| *dt/ $E_{max}$ <= t* | (algebra) |
| *dt/ $E_{max}$ <= t' - dt* | (2) |

*dt/ $E_{max}$ + dt <= t'*

# Relative error analysis

```
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:**

$dt/E_{max} + dt <= t'$

$.001/.05 + .05 <= t'$

$t' >= 0.070$ seconds (70 ms)

# Measurement pitfall #2: Unexpected cache effects

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

- **ip(array1, array2, array3);  /* 68,829 cycles */**
- **ipp(array1, array2, array3); /* 23,337 cycles */**

- **ipp(array1, array2, array3); /* 70,513 cycles */**
- **ip(array1, array2, array3);  /* 23,203 cycles */**

**Context switches can alter cache miss rate**

- **71,002  23,617  (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)
  - DCPI [Anderson, 1997] (performance counter interrupts)
  - SpeedShop [Zhaga, 1996] (performance counter interrupts)

# Profiling tools

| Tool | Overhead | Scope | Grain | Stalls |
|---|---|---|---|---|
| gprof | high | app | inst cnt | none |
| pixie | high | app | proc cnt | none |
| SimOS | high | sys | inst time | accurate |
| prof | low | app | inst time | none |
| DCPI | low | sys | inst time | accurate |
| SpeedShop | low | sys | inst time | inacurate |

***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:**

| cycles | % | procedure | image |
|---|---|---|---|
| 2064143 | 33.9 | ffb8ZeroPolyArc | /usr/shlib/X11/lib_dec_ffb_ev5.so |
| 517464 | 8.5 | ReadRequestFromClient | /usr/shlip/X11/libos.so |
| 305072 | 5.0 | miCreateETandAET | /usr/shlib/X11/libmi.so |
| 245450 | 4.0 | bcopy | /vmunix |
| 170723 | 2.8 | in_checksum | /vmunix |

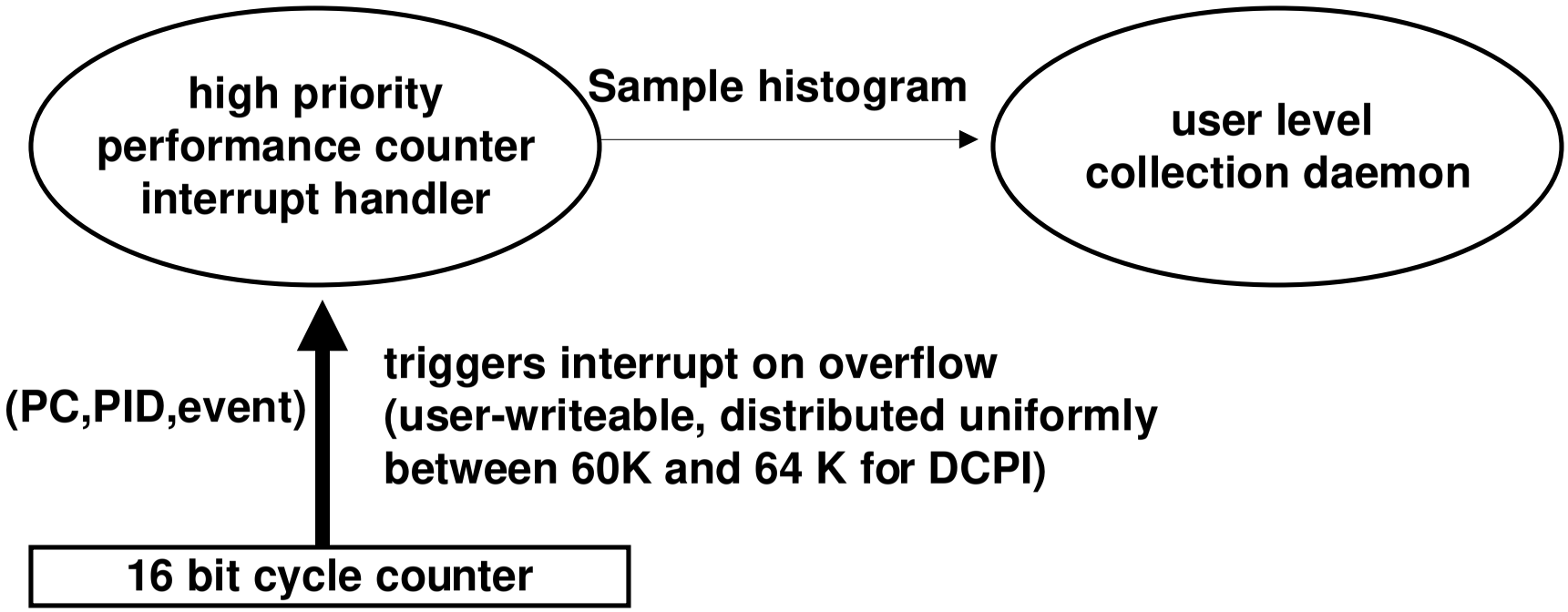
# DCPI case study

**Fine-grained profiling:**

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

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

# DCPI architecture



# 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.**

**For more info: Anderson, et al. "Continuous Profiling: Where Have all the Cycles Gone?", ACM TOCS, v15, n4, Nov. 1997, pp 357-390.**