Measuring Performance

Measurement tools and techniques

- Fundamental strategies
- Interval timers & cycle counters
- Program profiling
- Tracing
- Indirect measurement

Events

- Most measurement tools based on *events*
  - Some predefined change to system state
- Definition depends on metric being measured
  - Memory reference
  - Disk access
  - Change in a register's state
  - Network message
  - Processor interrupt

3

Event Classification

- ***Count*** metrics
  - The number of times event X occurs
  - Number of cache misses
  - Number of I/O operations

4

Event Classification

- ***Secondary-event*** metrics
  - Record a value when triggered by some event
  - Record block size for each I/O operation
  - Count number of operations
  - Find average I/O transfer size

5

Event Classification

- ***Profiles***
  - Characterization of overall behavior
  - Aggregate/big picture view of an application program
  - Time spent in each function

6

Event-Driven Strategies

- Record necessary information *only when selected event occurs*
- Modify system to record event
- Dump data when program terminates
  - May need intermediate dumps also
- E.g. simple counter in page fault routine

7

Event-Driven Strategies

- System overhead
  - Only when the event of interest actually occurs
  - Infrequent events → little perturbation
  - Frequent events → high perturbation
- No longer "typical" behavior?
  - Perturbation changes system being measured

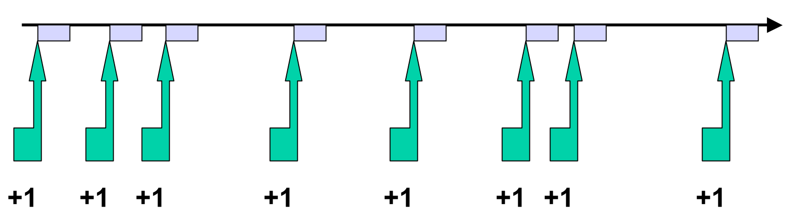
8

Event-Driven Strategies

- Inter-event time is unpredictable
  - Depends on when events actually occur
  - Makes it hard to estimate perturbation
  - How long to measure?
- Event-driven measurement tools
  - → Good for low-frequency events

9

Event-Driven Strategies



- Counts 8 events exactly

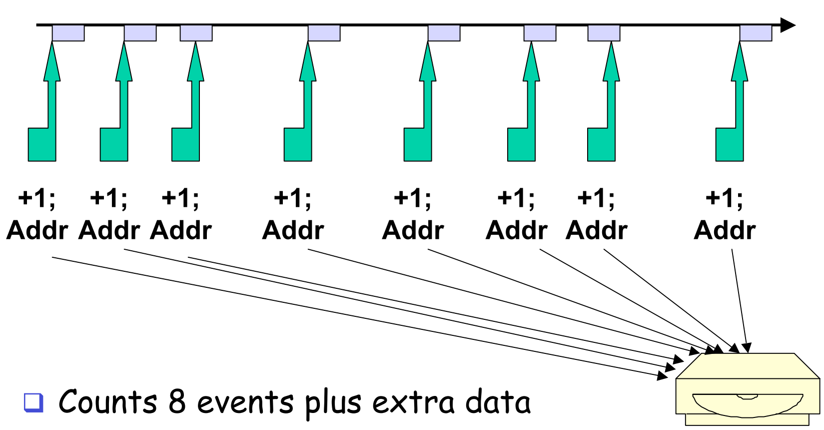
10

Tracing

- Similar to event-driven
- But record additional system state
  - Event has occurred – count
  - Additional information to uniquely identify event
  - E.g. addresses that cause page faults
- Overhead
  - Additional memory or disk storage
  - Time to save state
- Relatively large system perturbation

11

Tracing



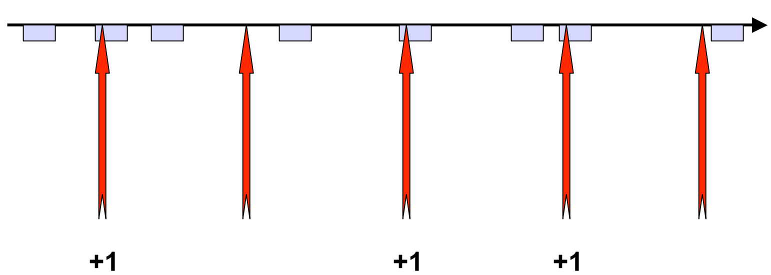
- Counts 8 events plus extra data

12

Sampling

- Record necessary state at fixed time intervals
- Overhead
  - Independent of specific event frequency
  - Depends on *sampling frequency*
- Misses some events
- Produces statistical summary
  - May miss infrequent events
  - Each replication will produce different results

Sampling



- Counts 3 events out of 5 samples

Comparisons

	Event count	**Tracing**	**Sampling**
Resolution	Exact count	Detailed info	Statistical summary
Overhead	Low	High	Constant
Perturbation	~ #events	High	Fixed

15

Comparison

- Event counting
  - Best for low frequency events
  - Required if exact counts needed
- Sampling
  - Best for high frequency events
  - If statistical summary is adequate
- Tracing
  - When additional detail is required

16

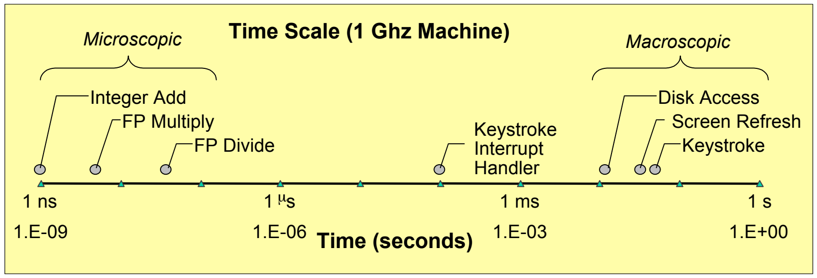
Indirect Measurements

- Used when desired metric is not directly accessible
- Measure one thing directly
  - Derive or deduce desired metric
- Highly dependent on creativity of performance analyst

Time Measurement

Based on Ch 9 of Computer Systems: A Programmer's Perspective - Bryant & O'Halloran

Computer Time Scales



- Two Fundamental Time Scales
  - Processor: ~10^{-9} sec.
  - External events: ~10^{-2} sec.
    - Keyboard input
    - Disk seek
    - Screen refresh
- Implication
  - Can execute many instructions while waiting for external event to occur
  - Can alternate among processes without anyone noticing

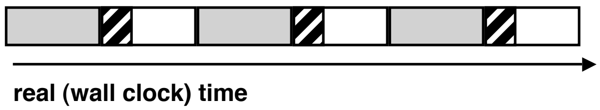
19

Measurement Challenge

- How Much Time Does Program X Require?
  - CPU time
    - How many total seconds are used when executing X?
    - Measure used for most applications
    - Small dependence on other system activities
  - Actual ("Wall") Time
    - How many seconds elapse between the start and the completion of X?
    - Depends on system load, I/O times, etc.
- Confounding Factors
  - How does time get measured?
  - Many processes share computing resources
    - Transient effects when switching from one process to another
    - Suddenly, the effects of alternating among processes become noticeable

20

"Time" on a Computer System



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

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

= **some other user's time** *(time executing instructions in different user's process)*

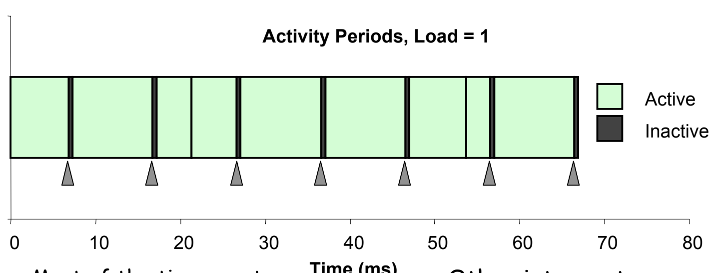
\+ + = **real (wall clock) time**

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

cumulative user time

21

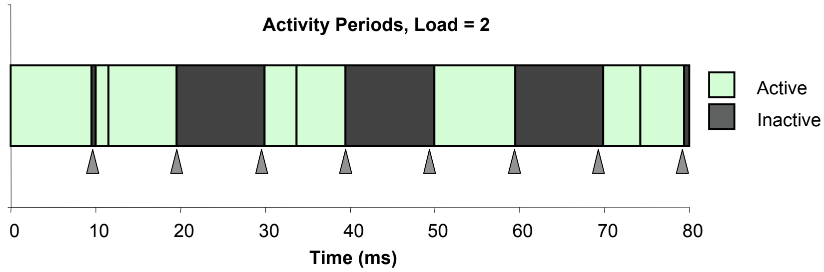
Activity Periods: Light Load



- Most of the time spent executing one process
- Periodic interrupts every 10ms
  - Interval timer
  - Keep system from executing one process to exclusion of others
- Other interrupts
  - Due to I/O activity
- Inactivity periods
  - System time spent processing interrupts
  - ~250,000 clock cycles

22

Activity Periods: Heavy Load

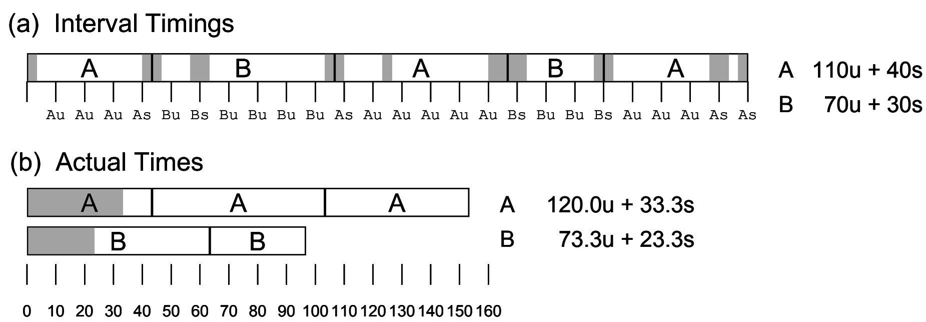


- Sharing processor with one other active process
- From perspective of this process, system appears to be "inactive" for ~50% of the time
  - Other process is executing

Interval Counting

- OS Measures Runtimes Using Interval Timer
  - Maintain 2 counts per process
    - User time
    - System time
  - Each time get timer interrupt, increment counter for executing process
    - User time if running in user mode
    - System time if running in kernel mode

Interval Counting Example



25

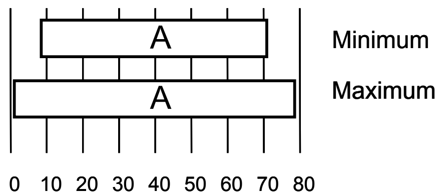
Unix `time` Command

```
time make osevent
gcc -O2 -Wall -g  -march=i486 -c clock.c
gcc -O2 -Wall -g  -march=i486 -c options.c
gcc -O2 -Wall -g  -march=i486 -c load.c
gcc -O2 -Wall -g  -march=i486 -o osevent osevent.c . . .
0.820u 0.300s 0:01.32 84.8%     0+0k 0+0io 4049pf+0w
```

- 0.82 seconds user time
  - 82 timer intervals
- 0.30 seconds system time
  - 30 timer intervals
- 1.32 seconds wall time
- 84.8% of total was used running these processes
  - (.82+0.3)/1.32 = .848

26

Accuracy of Interval Counting

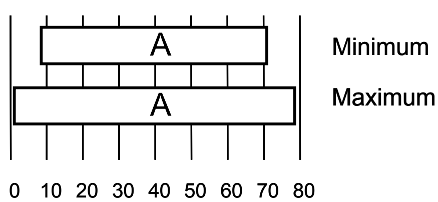


- **Computed time = 70ms**
- **Min Actual = 60 + ε**
- **Max Actual = 80 – ε**

- Worst Case Analysis
  - Timer Interval = δ
  - Single process segment measurement can be off by ±δ
  - No bound on error for multiple segments
    - Could consistently underestimate, or consistently overestimate

27

Accuracy of Int. Cntg. (cont.)



- **Computed time = 70ms**
- **Min Actual = 60 + ε**
- **Max Actual = 80 – ε**

- Average Case Analysis
  - Over/underestimates tend to balance out
  - As long as total run time is sufficiently large
    - Min run time ~1 second
    - 100 timer intervals
  - Consistently miss 4% overhead due to timer interrupts

28

Cycle Counters

- Most modern systems have built in registers that are incremented every clock cycle
  - Very fine grained
  - Maintained as part of process state
    - In Linux, counts elapsed global time
- 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

Cycle Counter Period

- Wrap Around Times for 550 MHz machine
  - Low order 32 bits wrap around every 2^{32} / (550 * 10^6) = 7.8 seconds
  - High order 64 bits wrap around every 2^{64} / (550 * 10^6) = 33539534679 seconds
    - 1065 years
- For 2 GHz machine
  - Low order 32-bits every 2.1 seconds
  - High order 64 bits every 293 years

Measuring with Cycle Counter

- Idea
  - Get current value of cycle counter
    - store as pair of unsigned's `cyc_hi` and `cyc_lo`
  - Compute something
  - Get new value of cycle counter
  - Perform double precision subtraction to get elapsed cycles

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

void start_counter()
{
  /* Get current value of cycle counter */
  access_counter(&cyc_hi, &cyc_lo);
}
```

31

Accessing the Cycle Cntr.

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

```
void access_counter(unsigned *hi, unsigned *lo)
{
  /* Get cycle counter */
  asm("rdtsc; movl %%edx,%0; movl %%eax,%1"
      : "=r" (*hi), "=r" (*lo)
      : /* No input */
      : "%edx", "%eax");
}
```

- Emit assembly with `rdtsc` and two `movl` instructions

32

Completing Measurement

- Get new value of cycle counter
- Perform double precision subtraction to get elapsed cycles
- Express as `double` to avoid overflow problems

```
double get_counter()
{
  unsigned ncyc_hi, ncyc_lo
  unsigned hi, lo, borrow;
  /* Get cycle counter */
  access_counter(&ncyc_hi, &ncyc_lo);
  /* Do double precision subtraction */
  lo = ncyc_lo - cyc_lo;
  borrow = lo > ncyc_lo;
  hi = ncyc_hi - cyc_hi - borrow;
  return (double) hi * (1 << 30) * 4 + lo;
}
```

33

Timing With Cycle Counter

- Determine Clock Rate of Processor
  - Count number of cycles required for some fixed number of seconds

```
  double MHZ;
  int sleep_time = 10;
  start_counter();
  sleep(sleep_time);
  MHZ = get_counter()/(sleep_time * 1e6);
```

- Time Function P
  - First attempt: Simply count cycles for one execution of P

```
  double tsecs;
  start_counter();
  P();
  tsecs = get_counter() / (MHZ * 1e6);
```

34

Measurement Pitfalls

- Overhead
  - Calling `get_counter()` 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:

```
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 */
```

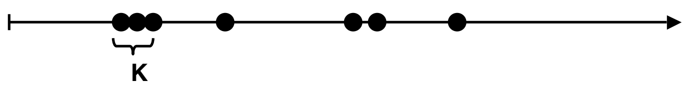

Dealing with Overhead & Cache Effects

- Always execute function once to "warm up" cache
- Keep doubling number of times execute P() until reach some threshold
  - Used CMIN = 50000

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

Multitasking Effects

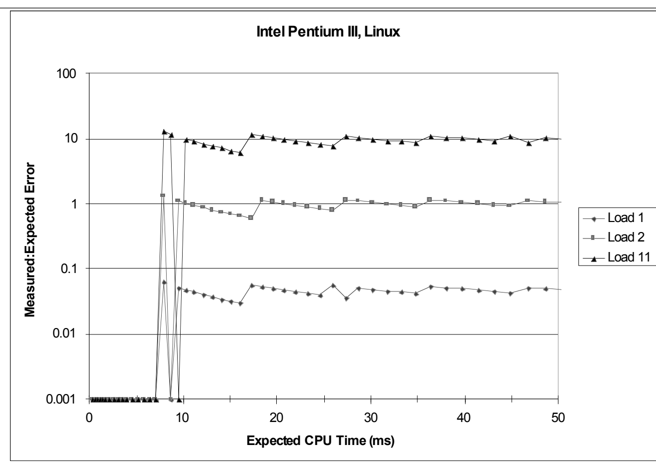
- Cycle Counter Measures Elapsed Time
  - Keeps accumulating during periods of inactivity
    - System activity
    - Running other processes
- Key Observation
  - Cycle counter never underestimates program run time
  - Possibly overestimates by large amount
- K-Best Measurement Scheme
  - Perform up to N (e.g., 20) measurements of function
  - See if fastest K (e.g., 3) within some relative factor ε (e.g., 0.001)



37

K-Best Validation

K = 3, ε = 0.001



- Very good accuracy for < 8ms
  - Within one timer interval
  - Even when heavily loaded
- Less accurate of > 10ms
  - Light load: ~4% error
    - Interval clock interrupt handling
  - Heavy load: Very high error

38

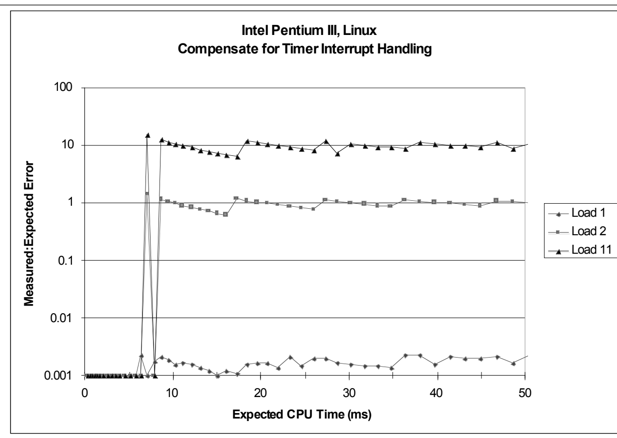
How are "actual" run times of programs determined?

- Write a procedure that repeatedly writes values to an array of 2048 integer and then reads them back
- Let r be the number of repetitions
- Determine expected run time T(r) of procedure as a function of r by timing it for r = 1…10 and performing a least squares fit to T(r) = mr + b
  - Linear regression (will discuss later this semester)

39

Compensate For Timer Overhead

K = 3, ε = 0.001

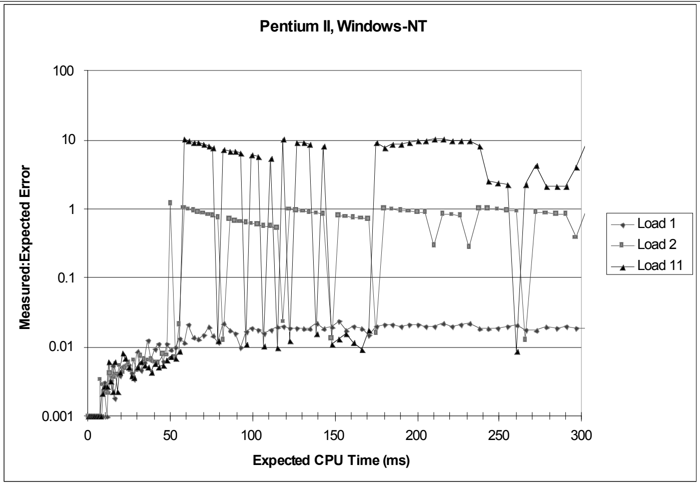


- Subtract Timer Overhead
  - Estimate overhead of single interrupt by measuring periods of inactivity
  - Call interval timer to determine number of interrupts that have occurred
- Better Accuracy for > 10ms
  - Light load: 0.2% error
  - Heavy load: Still very high error

40

K-Best on NT

K = 3, ε = 0.001



- Acceptable accuracy for < 50ms
  - Scheduler allows process to run multiple intervals
- Less accurate of > 10ms
  - Light load: 2% error
  - Heavy load: Generally very high error

41

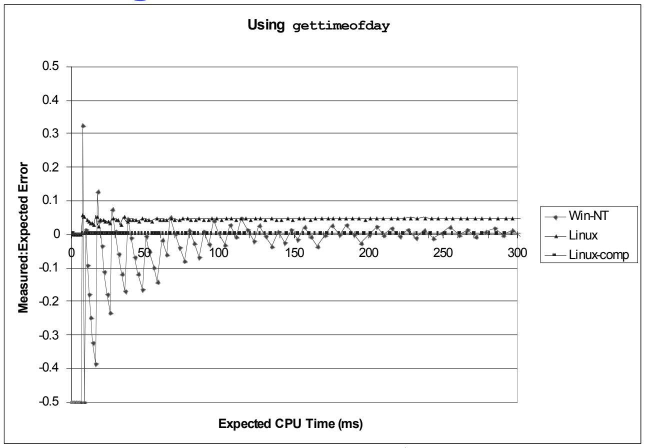
Time of Day Clock

- Unix `gettimeofday()` function
- Return elapsed time since reference time (Jan 1, 1970)
- Implementation
  - Uses interval counting on some machines
    - Coarse grained
  - Uses cycle counter on others
    - Fine grained, but significant overhead and only 1 microsecond resolution

```
#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);
```

42

K-Best Using `gettimeofday`



- Linux
  - As good as using cycle counter
  - For times > 10 microseconds
- Windows
  - Implemented by interval counting
  - Too coarse-grained

43

Measurement Summary

- Timing is highly case and system dependent
  - What is overall duration being measured?
    - > 1 second: interval counting is OK
    - << 1 second: must use cycle counters
  - On what hardware / OS / OS version?
    - Accessing counters
      - How `gettimeofday` is implemented
    - Timer interrupt overhead
    - Scheduling policy
- Devising a Measurement Method
  - Long durations: use Unix timing functions
  - Short durations
    - If possible, use `gettimeofday`
    - Otherwise must work with cycle counters
    - K-best scheme most successful

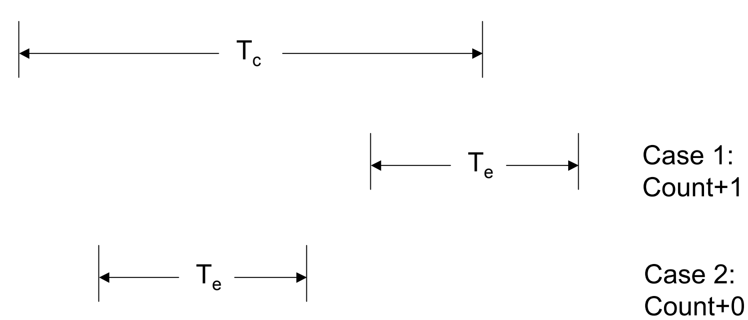
44

Approximate Measures of Short Intervals

- Suppose no access to cycle counters
- How to measure an event that is shorter than the resolution of the clock?
- Cannot directly measure events with $T_e < T_c$
- Overhead makes it hard to measure even when $T_e > nT_c$,
  - *n* is small integer

45

Approximate Measures of Short Intervals



46

Approximate Measures of Short Intervals

- Bernoulli experiment
  - Outcome = +1 with probability *p*
  - Outcome = +0 with probability (1-*p*)
  - Equivalent to flipping a biased coin
- Repeat *n* times
  - Approximates a binomial distribution
  - Only approximate since each measurement cannot be guaranteed to be independent
    - Usually close enough in practice

47

Approximate Measures of Short Intervals

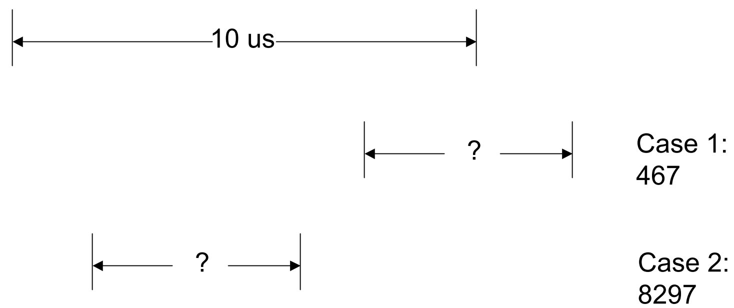
- *m* = number of times Case 1 occurs
  - Count+1
- *n* = total number of measurements
- Average duration is ratio of *m/n*
- Use confidence interval for proportions

$$T_e = \frac{m}{n} T_c$$

48

Example

- Clock resolution = 10 us
- n = 8764 measurements
- m = 467 clock ticks counted
- 95% confidence interval



Example

$$(c_1, c_2) = \frac{467}{8764} \mp 1.96 \sqrt{\frac{\frac{467}{8764}\left(1 - \frac{467}{8764}\right)}{8764}}$$

$$= (0.0486, 0.0580)$$

- Scale by clock period = 10 us
- 95% chance that measured event is
  - (0.49, 0.58) us

Important Aside

- Confidence interval calculation for proportions discussed in last class (and textbooks) is controversial
  - Recently, statisticians have shown that it is problematic
  - The approach used on the previous slide + in the textbooks (Lilja, Jain, others) is somewhat discredited
  - Link on class web page

51

Profiling

- Overall view of program's execution-time behavior
- Fraction of total time spent in specific states
  - Fraction of time in each subroutine
  - Fraction of time in OS kernel
  - Fraction of time doing I/O
- Find bottlenecks, code hot-spots
  - Optimize those sections first

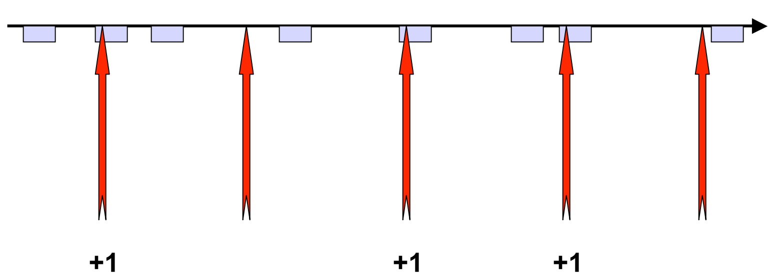
52

Statistical Sampling

- Select a random subset of a population
- Gather information on only this subset
- Extrapolate this information to overall population
- Results are a statistical summary with corresponding error probabilities

53

PC Sampling

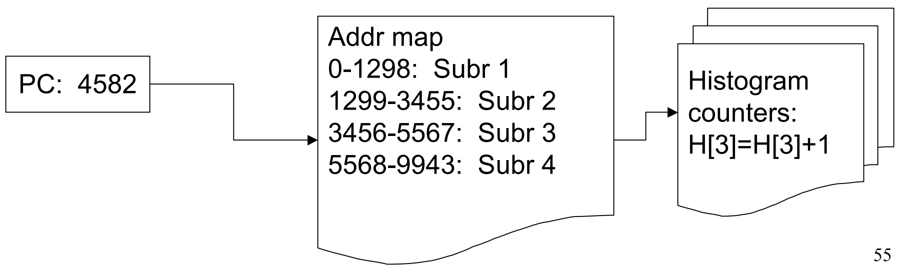


- Periodically interrupt program at fixed intervals
- Record appropriate state information in interrupt service routine
- Post-process to obtain overall profile

54

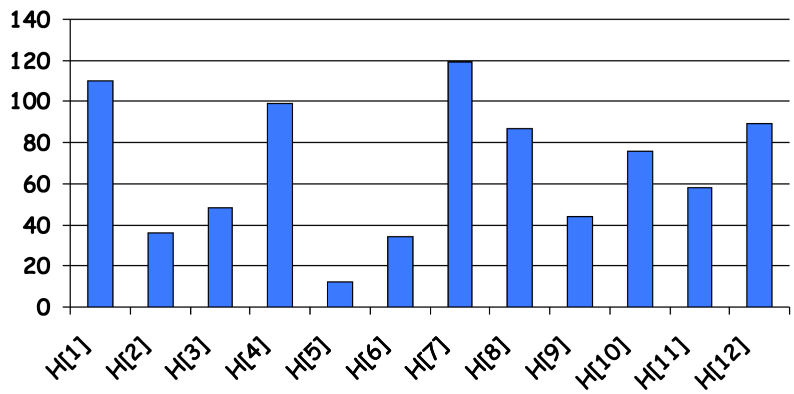
PC Sampling

- At each interrupt
  - Examine PC on return address stack
  - Use address map to translate this PC to subroutine i
  - Increment array element H[i]



55

PC Sampling



56

PC Sampling

- n total interrupts
- Post-processing step
  - H[i]/n = fraction of time executing in subroutine i
  - (H[i]/n) * (interrupt period) = time in each subroutine

57

PC Sampling

- This is a statistical process
  - Different counts each time the experiment is performed
- Infer behavior of entire program from small sample
- Apply confidence intervals to quantify precision of results

58

Example

- 40 us interrupt
- 36,128 interrupts in subroutine A
- Program runs for 10 seconds
- Time in this subroutine?
  - 90% confidence interval

- m = 36,128
- n = 10 sec / 40 us = 250,000
- p = m/n = 0.144

59

Example

$$(c_1, c_2) = 0.144512 \mp 1.645 \sqrt{\frac{0.144512(0.855488)}{250000}}$$

$$= (0.144, 0.146)$$

- 90% chance that the program spent 14.4-14.6% of its time in subroutine A

60

Example

- 10 ms interrupt
- 12 interrupts in subroutine A
- n = 800 samples
  - 8 seconds total execution time
- Time in this subroutine?
  - 99% confidence interval

- p = m/n = 0.015

Example

$$(c_1, c_2) = 0.015 \mp 2.576\sqrt{\frac{0.015(1-0.015)}{800}}$$

$$= (0.0039, 0.0261)$$

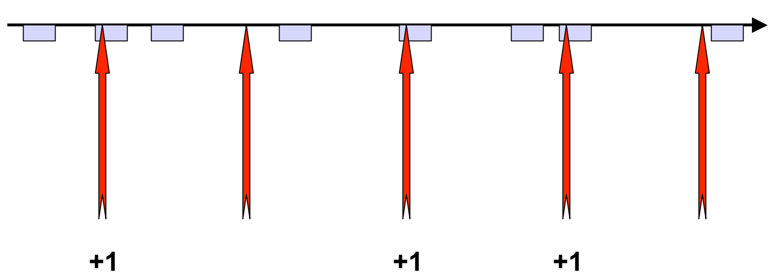
- 99% chance that the program spent 31-210 ms in subroutine A
- A pretty wide range!
- But only <3% of total execution time
- Start optimizing somewhere else first

Reducing the Interval Size

- Use a lower confidence level
- Obtain more samples
  - Run program longer
    - May not be possible
  - Increase sample rate
    - May be fixed by system
    - Will increase overhead and perturbation
  - Run multiple times and add samples from each run

63

PC Sampling



- Interrupts must occur asynchronously w.r.t. any program events
  - Samples must be independent of each other
  - Else over/under-sample events synchronous with interrupt
- Periodic versus random sampling

64

Basic Block Counting

- Basic block
  - *Sequence of instructions with no branches into or out of the block*
  - When first instruction is executed, guaranteed that all instructions in block will be executed
  - Single entry, single exit

Basic Block Counting

- Generate a program profile by inserting additional instructions in each block
  - Increment a unique counter each time a block is entered
- Produces a histogram of program execution
- Can post-process to find instruction execution frequencies

Comparison

	PC sampling	Basic block counting
Output	Statistical estimate	Exact count
Overhead	Interrupt service routine	Extra instructions per block
Perturbation	Randomly distributed	High
Repeatability	Within statistical variance	Perfect

67

Profiling Tools

- UNIX gprof
  - Uses PC-sampling
- Intel VTUNE
- Apple Shark
- Many others…

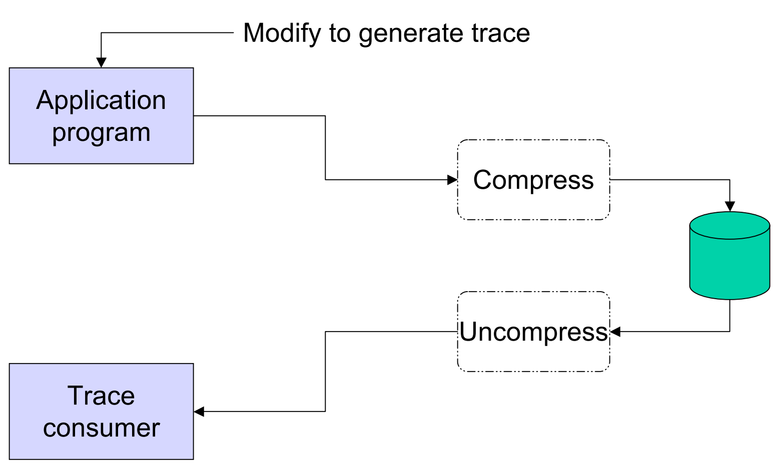
68

Event Tracing

- Profile shows overall *frequency-of-execution* behavior
  - Ignores time-ordering of events
- Program trace
  - Dynamic list of events generated by program
  - *Events* = anything you want to instrument
    - Sequence of memory addresses
    - I/O blocks accessed
- Typically used to drive a simulator

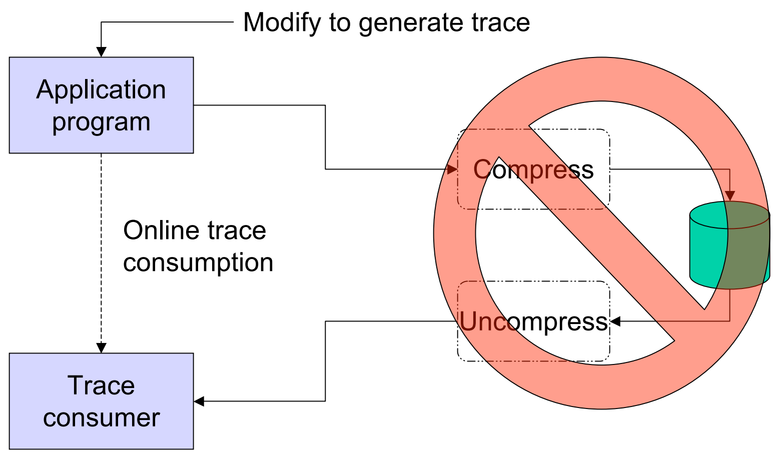
69

Trace Generation



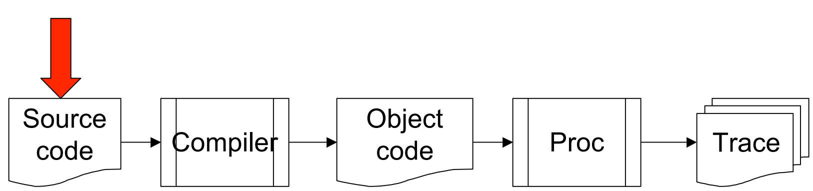
70

Trace Generation



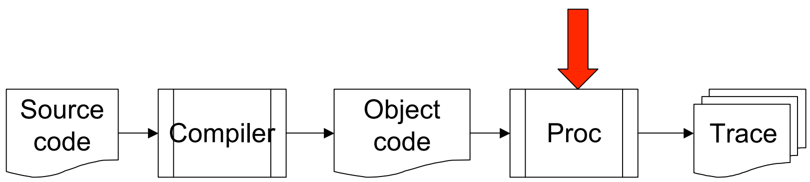
Trace Generation

- Source-code modification
  - Allows precise control of what events are traced and what data is recorded
  - Typically a manual process



Trace Generation

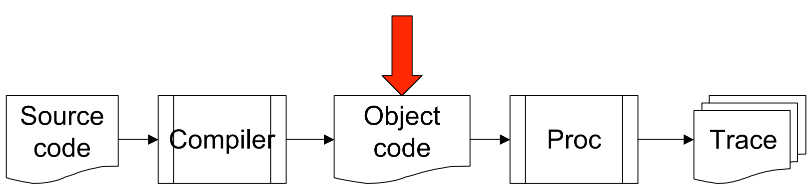
- Software exceptions
  - HW forces an exception before each instruction
  - Exception routine decodes instruction
    - Store instr type, PC, operand addresses, etc.
  - "Trace" bit in many processors
  - Tremendous slowdown



73

Trace Generation

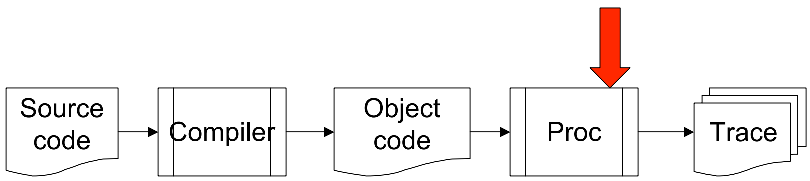
- Emulation
  - Make a system appear to be something else
  - Modify emulator to generate trace
  - E.g. Java Virtual Machine



74

Trace Generation

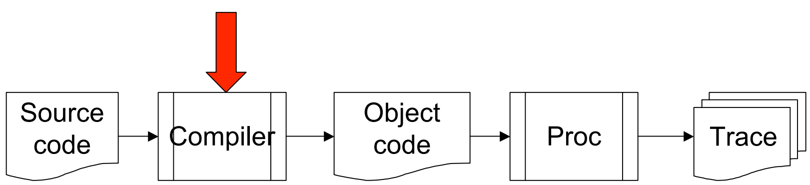
- Microcode modification
  - Modify instruction execution directly
  - Allows tracing of *all* instructions
    - Including operating system
  - Depends on access to lower levels of the processor
  - E.g. Transmeta Crusoe processor



75

Trace Generation

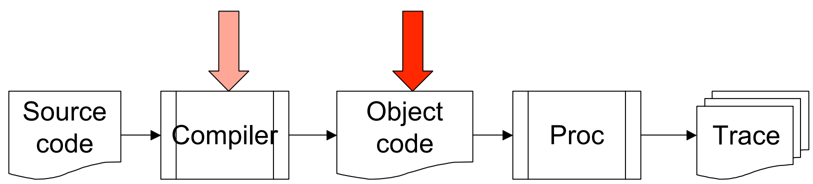
- Compiler modification
  - Insert trace code directly in object file
  - Requires access to the compiler itself



76

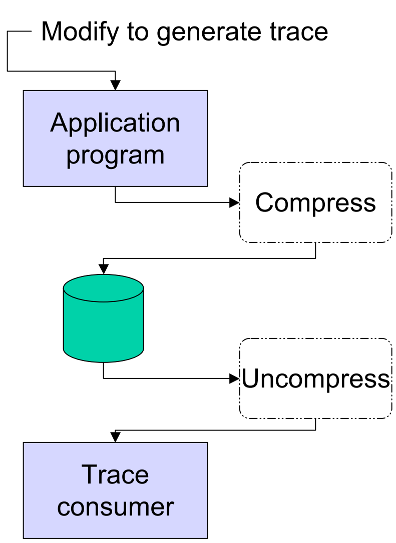
Trace Generation

- Compiler modification
  - Insert trace code directly in object file
  - Requires access to the compiler itself
  - Write post-compilation binary editor/rewrite tool



77

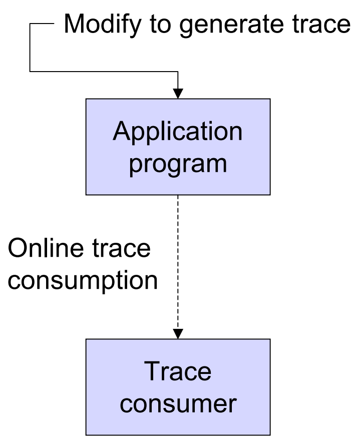
Trace Data Compression



- Standard compression algorithms as trace is written to disk
- Uncompress when reading
- Typical reduction
  - 20-70%
- Tradeoff is compress-uncompress time

78

Online Trace Consumption



- Use trace data as it is generated
- Never stored on disk
- Multitasking may lead to non-deterministic behavior
  - Repeatability issue
- Before-and-after comparison tests
  - Difference due to change in system or change in trace?
  - Becomes statistical comparison with *n* runs

79

Trace Data

- Tracing generates a tremendous volume of data
- Trace 100,000,000 instrs/sec
- 16 bits of data per event
- 190 Mbytes of data per second
  - 11 Gbytes per minute
- Huge perturbations
  - Due to tracing code
  - Time to store trace data

80

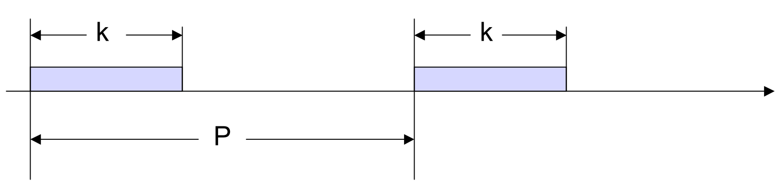
Advanced Techniques

- Researchers have developed many approaches to dealing with voluminous trace data
  - Abstract Execution
  - Trace Sampling
  - .....
- See Lilja

81

Trace Sampling

- Save only subsequences of overall trace
- Drive simulator with samples
- Results should be statistically similar to driving with complete trace
- One sample = *k* consecutive events
- Sampling interval = *P* (period)



82

Indirect *Ad Hoc* Techniques

- Sometimes the desired metric cannot be measured directly
- Use your creativity to measure one thing and then derive/infer the desired value

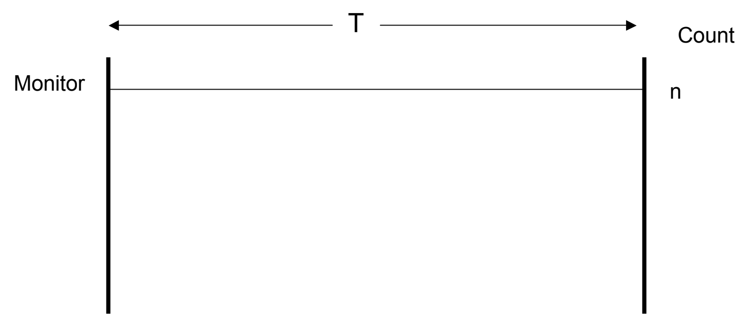
83

Example 1 – System Load

- What is system load?
  - Number of jobs in run queue?
  - Number of jobs actively time-sharing?
  - Fraction of time processor is not in idle loop?
  - Others?
- How to measure it?
  - Modify OS
  - PC sampling
  - Indirect?

84

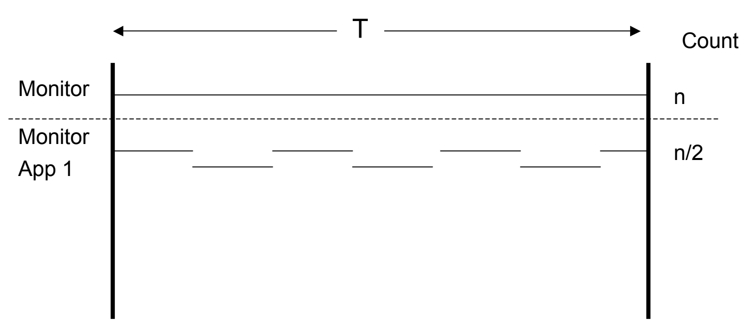
Example



- Let system run for fixed time *T*
- Note value of counter

85

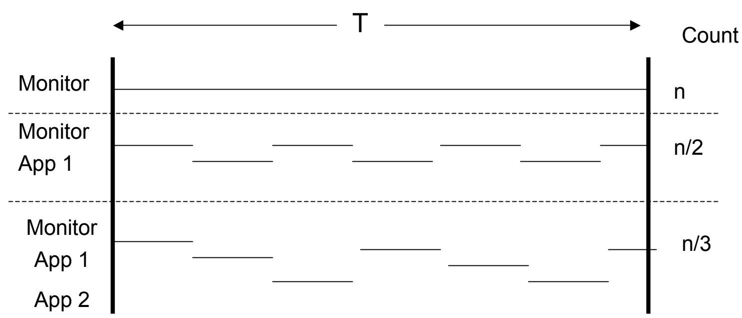
Example



- Let system run for fixed time *T*
- Compare value of loaded system monitor counter to unloaded system count value

86

Example



- Let system run for fixed time *T*
- Compare value of loaded system monitor counter to unloaded system count value

87

Example 2: The Memory Mountain

- Read throughput (read bandwidth)
  - Number of bytes read from memory per second (MB/s)
- Memory mountain
  - Measured read throughput as a function of spatial and temporal locality.
  - Compact way to characterize memory system performance.

88

Memory Mountain Test Function

```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                     /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

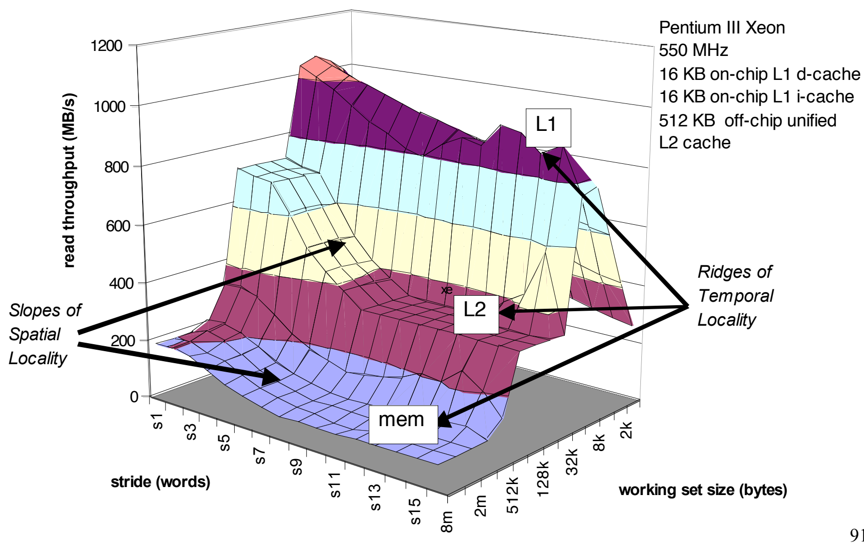
89

Memory Mountain Main Routine

```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;        /* Working set size (in bytes) */
    int stride;      /* Stride (in array elements) */
    double Mhz;      /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);              /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}
```

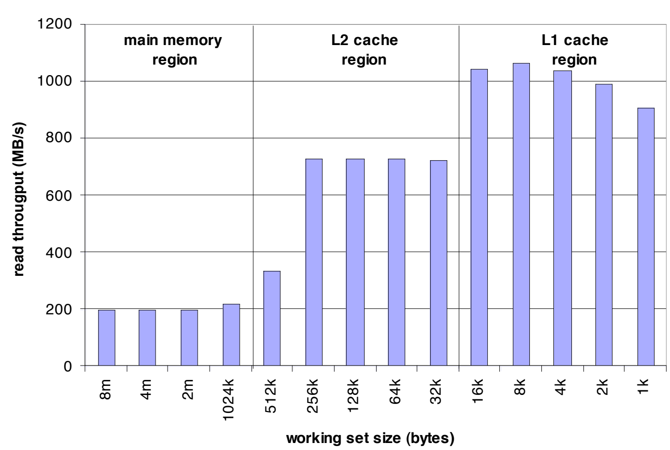
90

The Memory Mountain



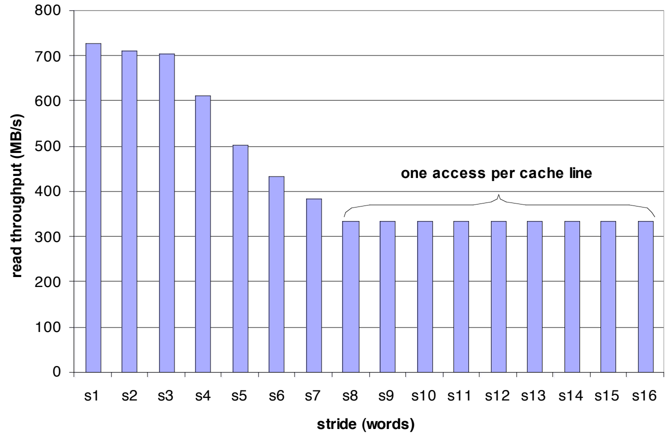
Ridges of Temporal Locality

- Slice through the memory mountain with stride=1
  - illuminates read throughputs of different caches and memory



A Slope of Spatial Locality

- Slice through memory mountain with size=256KB
  - shows cache block size.



93

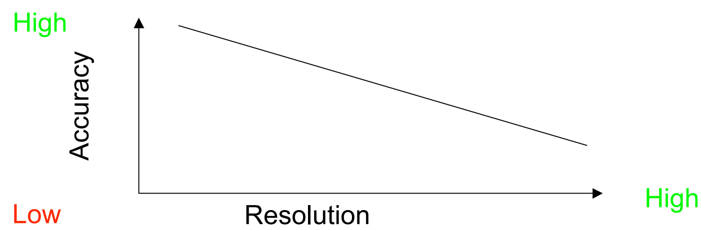
Perturbation

- To obtain more information (higher resolution)
  - → Use more instrumentation points
- More instrumentation points
  - → Greater perturbation

94

Perturbation

- Computer performance measurement uncertainty principle
  - ***Accuracy is inversely proportional to resolution.***



95

Perturbation

- Superposition does not work here
  - Non-linear
  - Non-additive
- Double instrumentation ≠ double impact on performance
  - Some instrumentation cancels out
  - Some multiplies impact
- No way to predict!

96

Instrumentation Code

- Changes memory access patterns
  - Affects memory banking optimizations
- Generates additional load/store instructions
  - More frequent cache flushes and replacements
  - But may reduce set associativity conflicts
- Generates more I/O operations
- Will increase overall execution time
  - More time-sharing context switches
- Alters virtual memory paging behavior

97

Summary

- Measurement strategies
  - Event-driven
  - Tracing
  - Sampling
- Measuring program time
- Profiling
- Trace generation
- Indirect measurements when all else fails
  - System load example
- Perturbations
  - Have to be careful to minimize perturbations due to instrumentation

98