Experimental Approaches in Computer Science

Dror Feitelson
Hebrew University

Lecture 4 – Microbenchmarks

What would you like to measure?

- Overhead of context switch
- Overhead of trap into kernel
- Memory bandwidth
- Network bandwidth
- CPU frequency
- Cache size

time for something

rate of something

system parameters

Microbenchmark – a benchmark designed to measure a specific feature

- Time
  - using gettimeofday()
- Rate
  - is work per unit time
  - so enough to measure time again
- Parameters
  - can often be inferred from discontinuities in timing measurements

lmbench

McVoy and Staelin

Usenix Technical Conf, Jan 1996

lmbench is a microbenchmark suite for computer systems, with emphasis on low-level primitives

Goals:

- Focus on basic building blocks used in system design
- Compare systems from different vendors
- Portability by using common tools

Memory bandwidth

- Size = 8MB to defeat caches but fit into memory
- bcopy gives half the bandwidth of read/write, because does both
- For small sizes could be 1/3 of the bandwidth, because destination cache lines need to be read first before being partially overwritten
- For read suggest to sum up the read data to enable optimization but avoid losing the whole operation
  - Memory access much higher than add, so not a large perturbation

IPC bandwidth

- Pipes: transfer 50MB in 64KB chunks
  - attempt to reduce effect of OS and context switching
- TCP: use 1MB chunks, 1MB buffers, loopback mode
  - attempt to get optimal performance
- These configurations actually based on memory copy, so should be related to memory bandwidth
  - may show use of optimizations to reduce copying
  - may depend on chunk and buffer sizes

Memory latency

- Def 1: time for a single cache miss
  - Reflects best achievable performance
  - Hard to measure in software
- Def 2: time for one in a sequence of cache misses with dependencies
  - Possible to measure in software
  - Better reflects effect on real applications:

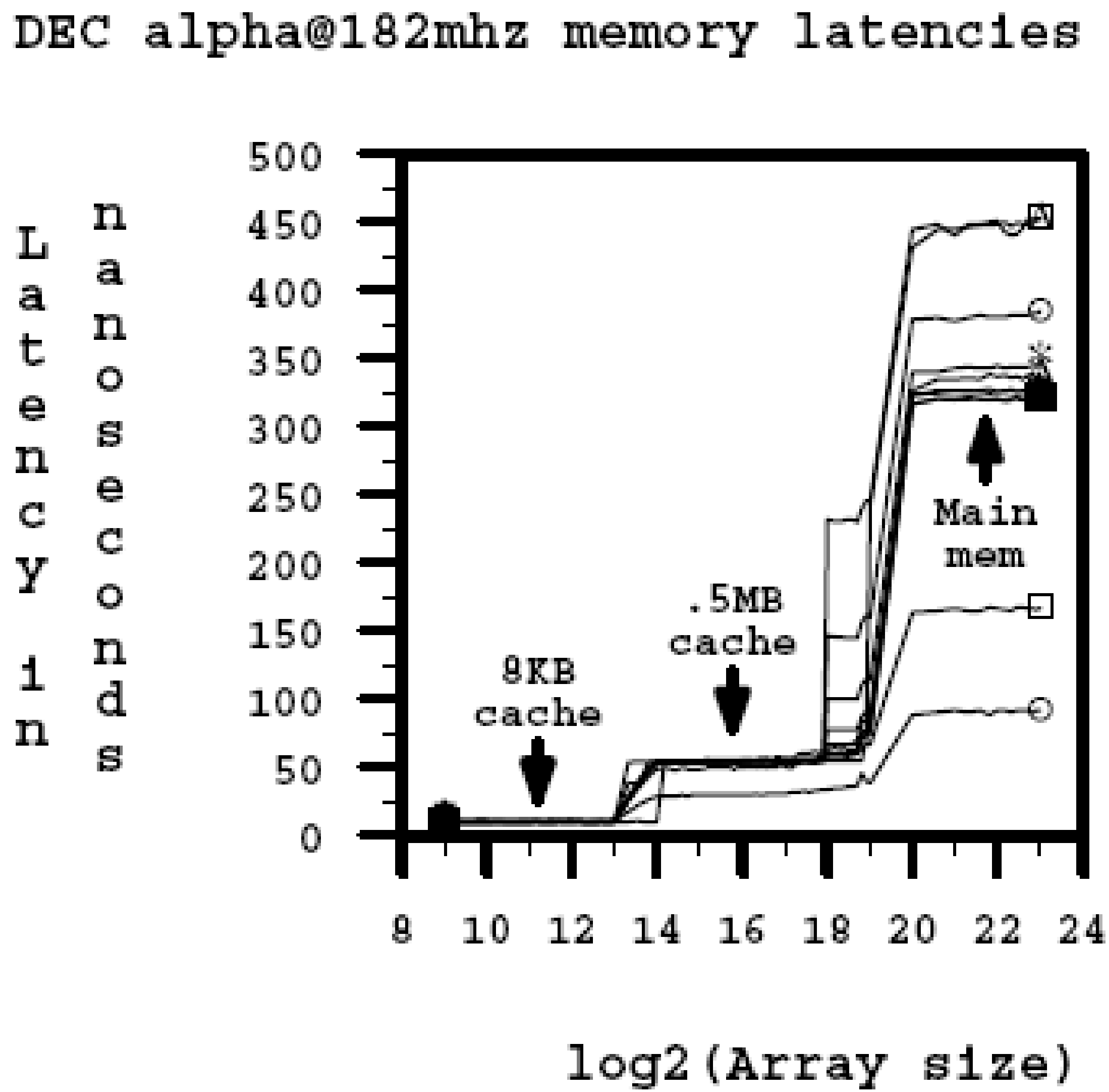
```
;p = head
(while (p -> next
;p = p->next
```

  - Loop overhead can be 100 times less than access

The actual benchmark:

- Array of size *n*
- Cells contain address of cell that is *k* away (wrap back at end)
- Walk the array using `p = *p;`
- Do this for different *n* and *k* (powers of 2)

- Identify cache and memory sizes by steps in graph
- Identify cache line size by smallest stride in main batch at next level (those that are faster benefit from multiple hits in same line)



Entry into the operating system

- Suggest using a loop of writing a single byte to /dev/null
- This is not optimized away in any system
- Alternatives like getpid or gettimeofday may be optimized or implemented as a user-level library function

Process creation alternatives

- Do fork and wait, child immediately exits
- Do fork and wait, child execs a hello world program
  - more realistic use
- Do fork and wait, child uses shell to run a hello world program
  - include searching $PATH

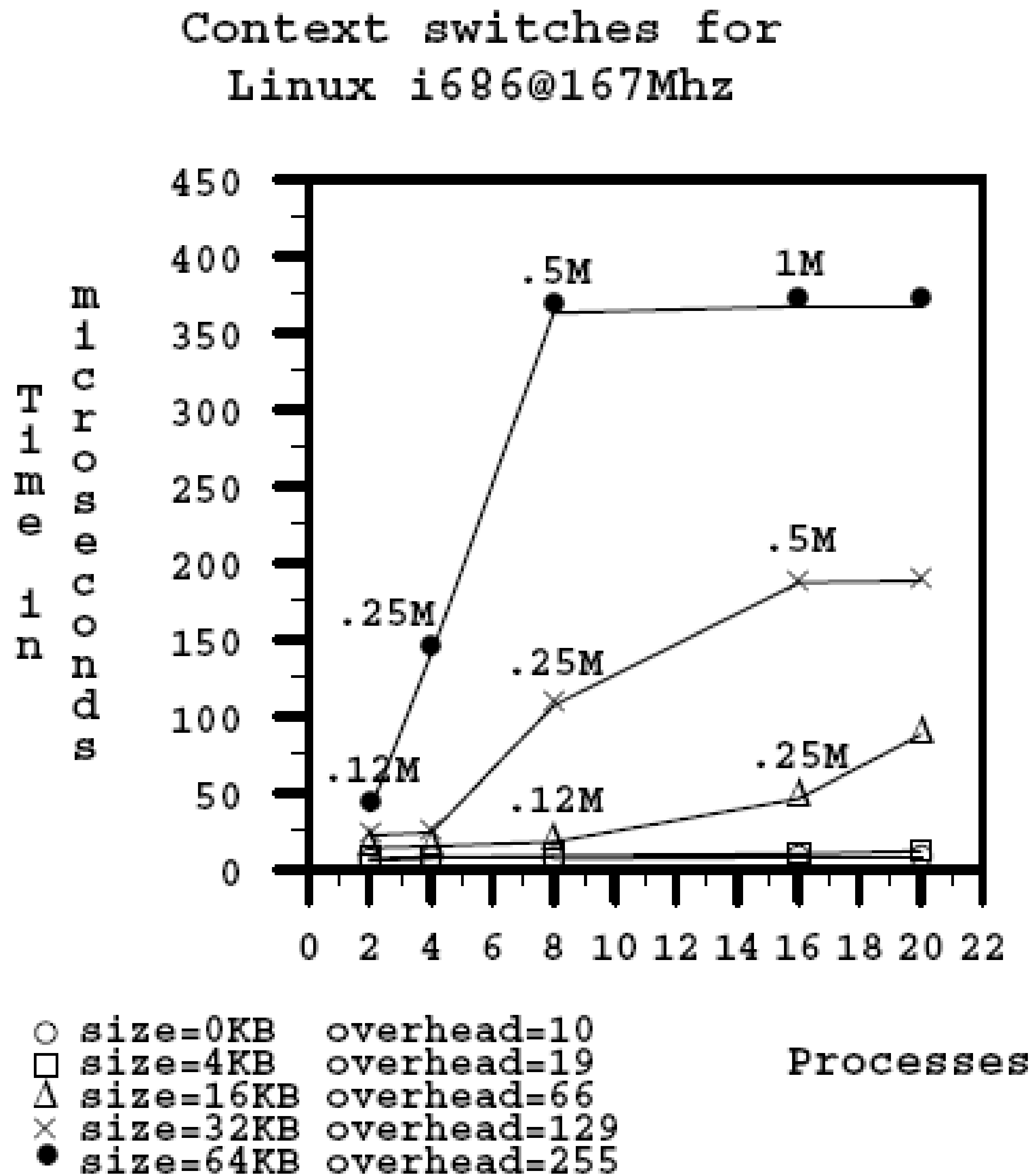
Context Switch

- Ousterhout [1991]: create two processes that pass a byte back and forth via a pipe
- Problems:
  - Overhead to read/write pipe is high and varied
  - Only two processes
  - Processes do not have any working set, so effect on cache is missed

Context Switch

- lmbench: create 2-20 processes that pass a token in a loop via pipes
  - Measure 2000 transfers of the token
  - Each process sums an array in memory before forwarding the token; repeat for different array sizes
  - Also do this on a single process to subtract overheads for read/write and summing from a hot cache

- Concentration of values at bottom left shows caching is effective across context switches
- No increase in latency as long as all working sets together fit into L2 = 256K



mhz benchmark

Staelin and McVoy

Usenix Technical Conf, Jan 1998

MHz benchmark: what is the clock rate on your machine?

- Idea: measure the time of *k* instructions, and divide by *k*
- Problems:
  - Low resolution for measuring this time
  - *k* C instructions can be compiled into a different number of machine instructions
  - On superscalar out-of-order processors operations may overlap

- Inspiration: in the 19^{th} century, chemists and physicists found the atomic weight of the elements by finding the greatest common divisor of a set of measurements
- Similarly, the cycle time of a computer is the greatest common divisor of the times needed to complete a set of different instructions
- Only assumption: every instruction takes an integral number of clock ticks
- Requirement: find instructions that take relatively prime numbers of cycles

Finding the GCD

- Problems
  - The measured times are not integral
  - The measurements include noise
- Solution
  - Let e_{min} be the smallest measurement
  - For i=1..6, calculate $b_i = e_{min} / i$ (these are candidates for being the cycle time)
  - Turn each measurement e_j into cycles by $c_j=[e_j/b_i]$
  - Check whether (e_j,c_j) fit a straight line through (0,0)
  - The i that gives the best fit is chosen

Example:

$e_1=6.9$

$e_2=10.6$

$e_3=17.7$

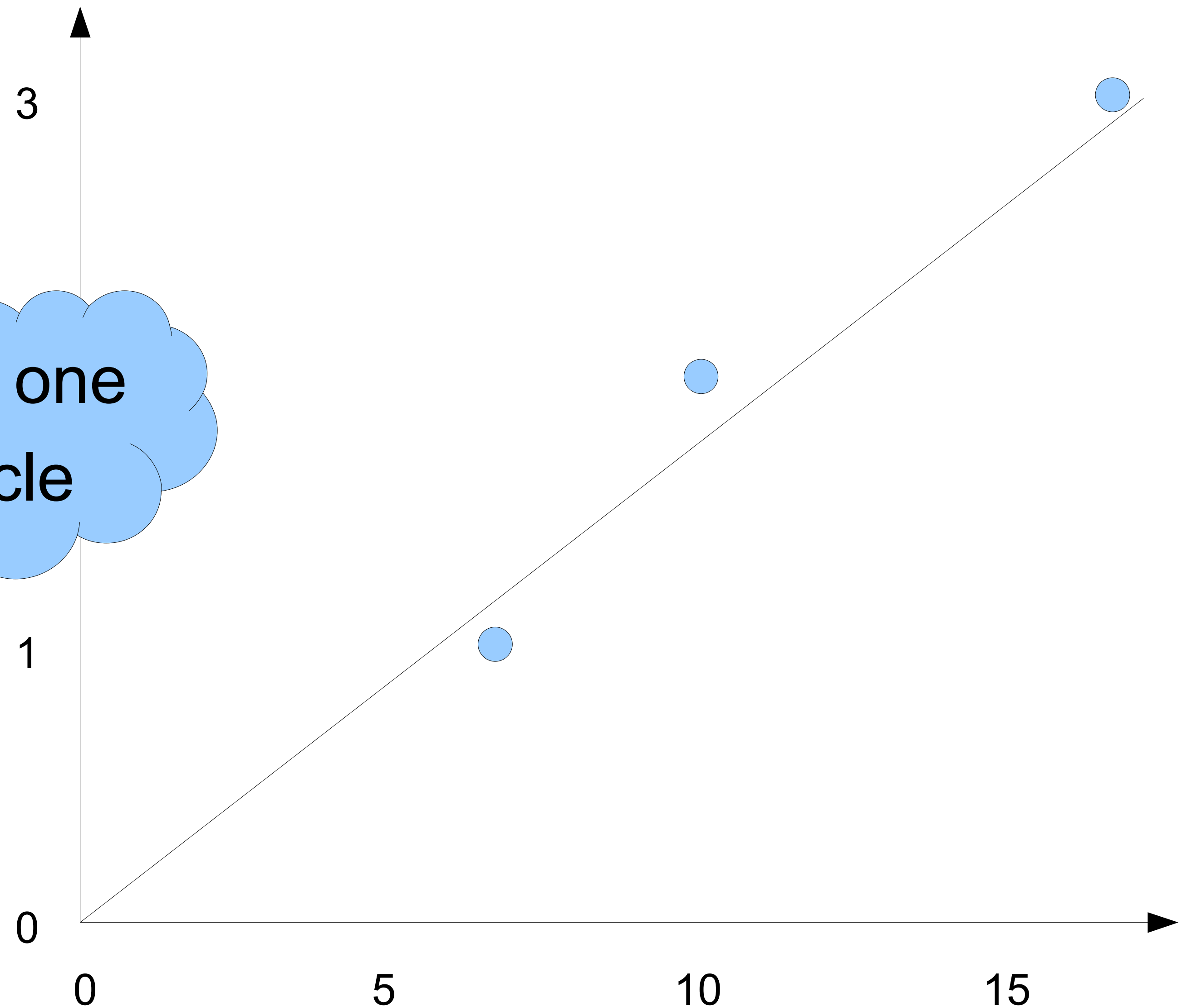
$i=1$

e_1 is one cycle

$c_1=1$

$c_2=[1.536]$

$c_3=[2.565]$



Example:

e_1=6.9

e_2=10.6

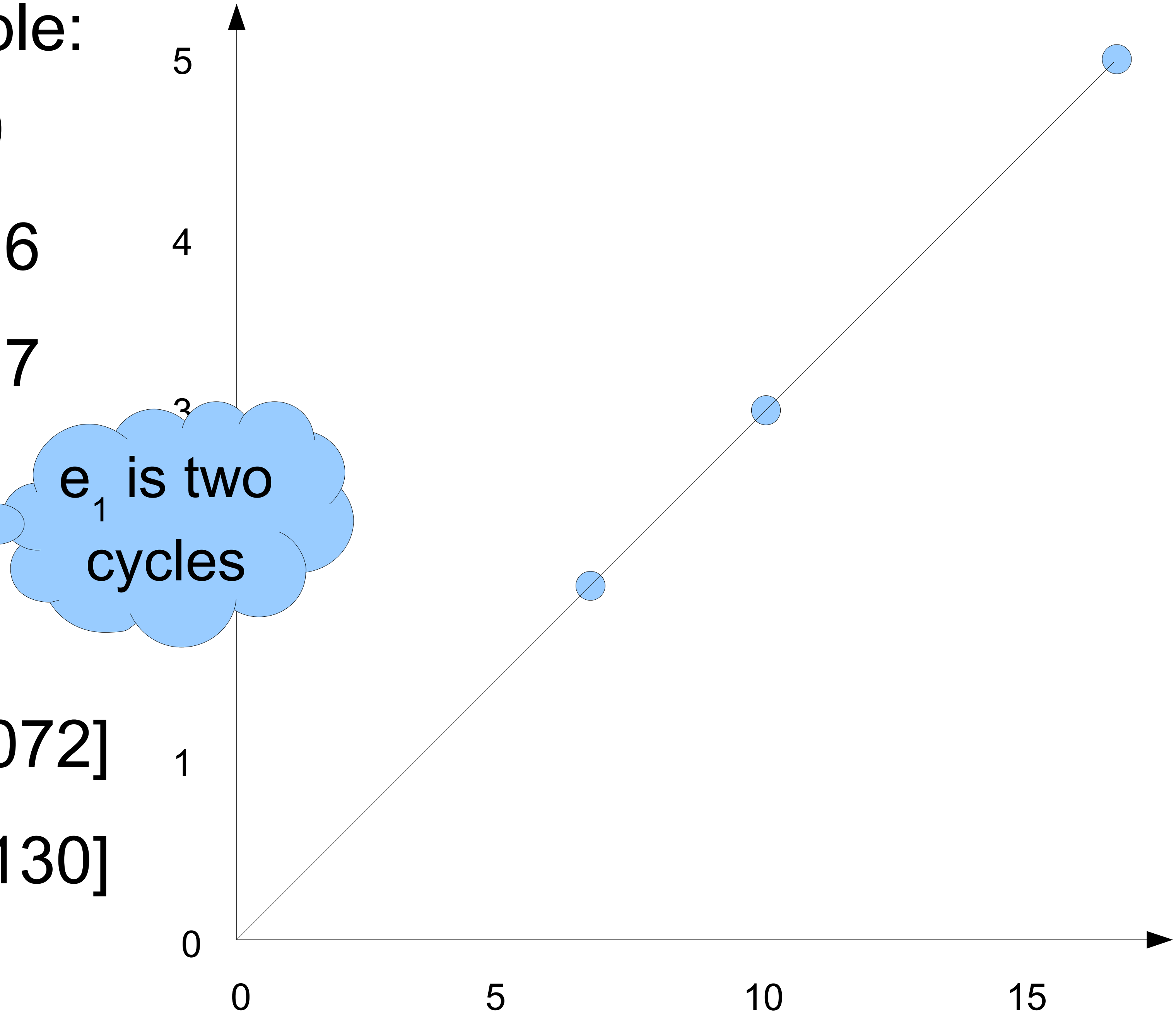
e_3=17.7

i=2

c_1=2

c_2=[3.072]

c_3=[5.130]



Atomic instructions

- Will string together 100 times for a measurement
- Requirements:
  - Each depends on the previous one so will not be done in parallel
  - Even subexpressions cannot be done in parallel
  - Compiler cannot optimize them away

Compiler optimization problems

- Instruction: a += a
- Optimized to a = 0
  - a += a is equivalent to a = a<<1
  - Repeated 100 times this is a = a << 100
  - But a only has 32 bits
  - So the whole 100 repetitions are replaced by one instance of a=0

The selected instructions:

```
;p = *p
;a ^= a + a
;a ^= a + a + a
;a >>= b
;a >>= a + a
;a ^= a << b
;a ^= a + b
a += (a + b) & 07
;a++;  a ^= 1;  a <<= 1
```

Several pairs are the same except for one additional operations

hopefully turns into one additional cycle

Summarizing repetitions of a measurement

- Repetitions usually lead to different results
- Some of the results are very different (outliers)
- Others just reflect uncertainty in the measurement (noise)
- How do we turn such multiple measurements into a single estimate?

Simple answer: take the average

- Average reflects all the measurements
- Minimizes $\sum (x_i - m)^2$

Which average?

- Arithmetic average
- Harmonic average
- Geometric average

Arithmetic average $\bar{x} = \frac{1}{n}\sum x_i$

- Good for measured times
- When measured times double, so does the average

Harmonic average $\bar{x} = \dfrac{1}{\frac{1}{n}\sum \frac{1}{x_i}}$

- Good for measured rates

$$x_i = w / t_i \quad \rightarrow \quad \bar{x} = \frac{n\,w}{\sum t_i}$$

total work

total time

- When measured times double, the average should be halved

Geometric average $\bar{x} = \sqrt{\prod x_i}$

- Gives consistent results when all x_i are measured relative to one of them, all have same weight
  - Therefore used in SPEC
- However, inconsistent with total time
  - If times double, average does not
- Useful for average of multiplicative process
  - X_i is improvement factor of component i
  - Average improvement of all components given by geometric mean

Measurement results

	Benchmark 1	Benchmark 2
System A	13 sec	16.5 sec
System B	19.5 sec	11 sec

Normalized by system A

	Benchmark 1	Benchmark 2	average
System A	1	1	1
System B	1.5	0.667	1.08

Normalized by system B

	Benchmark 1	Benchmark 2	average
System A	0.667	1.5	1.08
System B	1	1	1

Geometric average $\bar{x} = \sqrt{\prod x_i}$

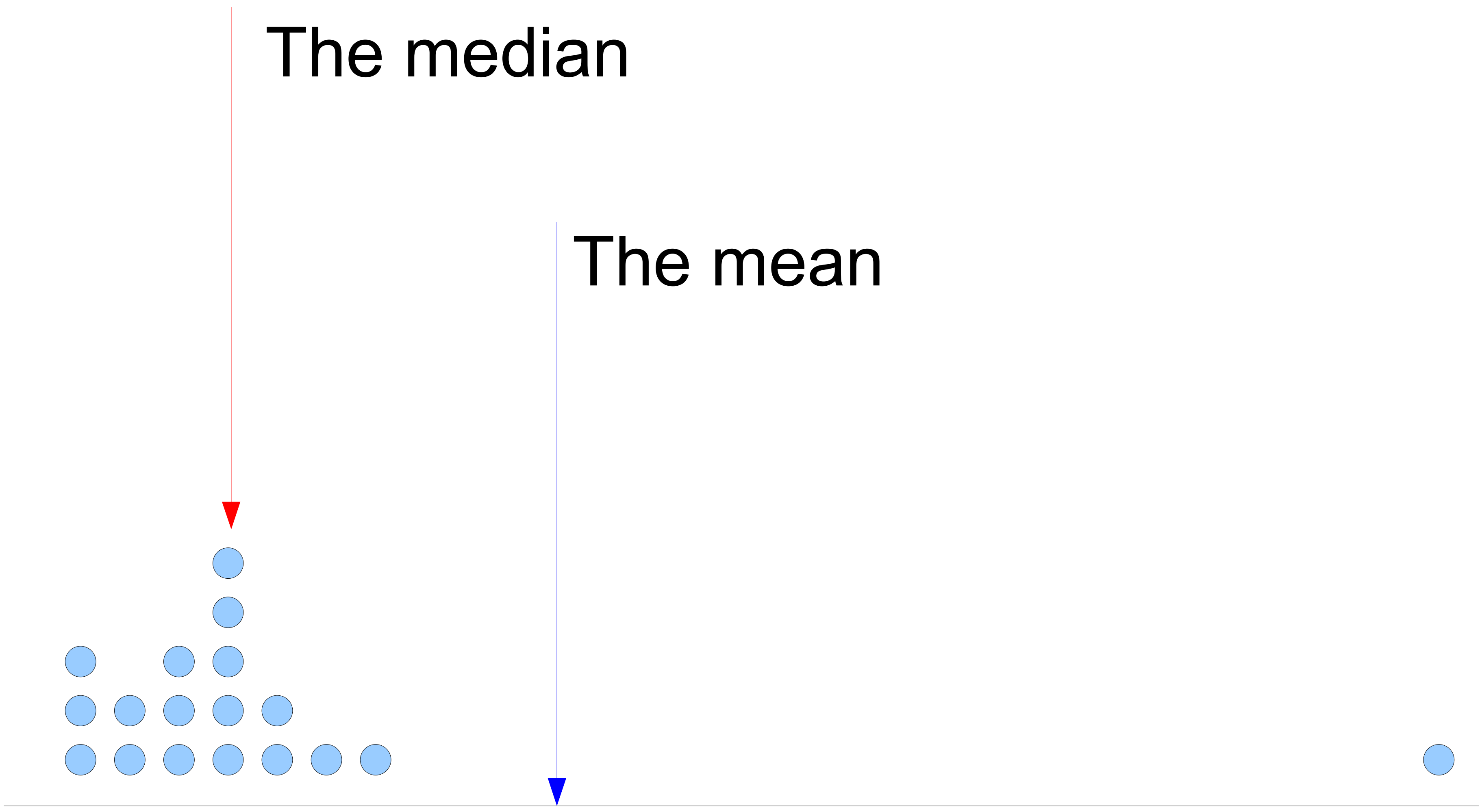
- Gives consistent results when all x_i are measured relative to one of them, all have same weight
  - Therefore used in SPEC
- However, inconsistent with total time
  - If times double, average does not
- Useful for average of multiplicative process
  - X_i is improvement factor of component i
  - Average improvement of all components given by geometric mean

- Alternative 1: the median

- More robust in face of outliers

- Minimizes $\sum \left| x_i - m \right|$

Example:



- Alternative 2: use the minimal value

- Interference typically adds time to the measurement
- So the minimal measurement is the one that has suffered the least noise
- Potential problem: if subtracting measurement overhead, minimal result may actually reflect subtraction of an inflated overhead measurement

Using microbenchmarks to analyze system performance

Brown & Seltzer

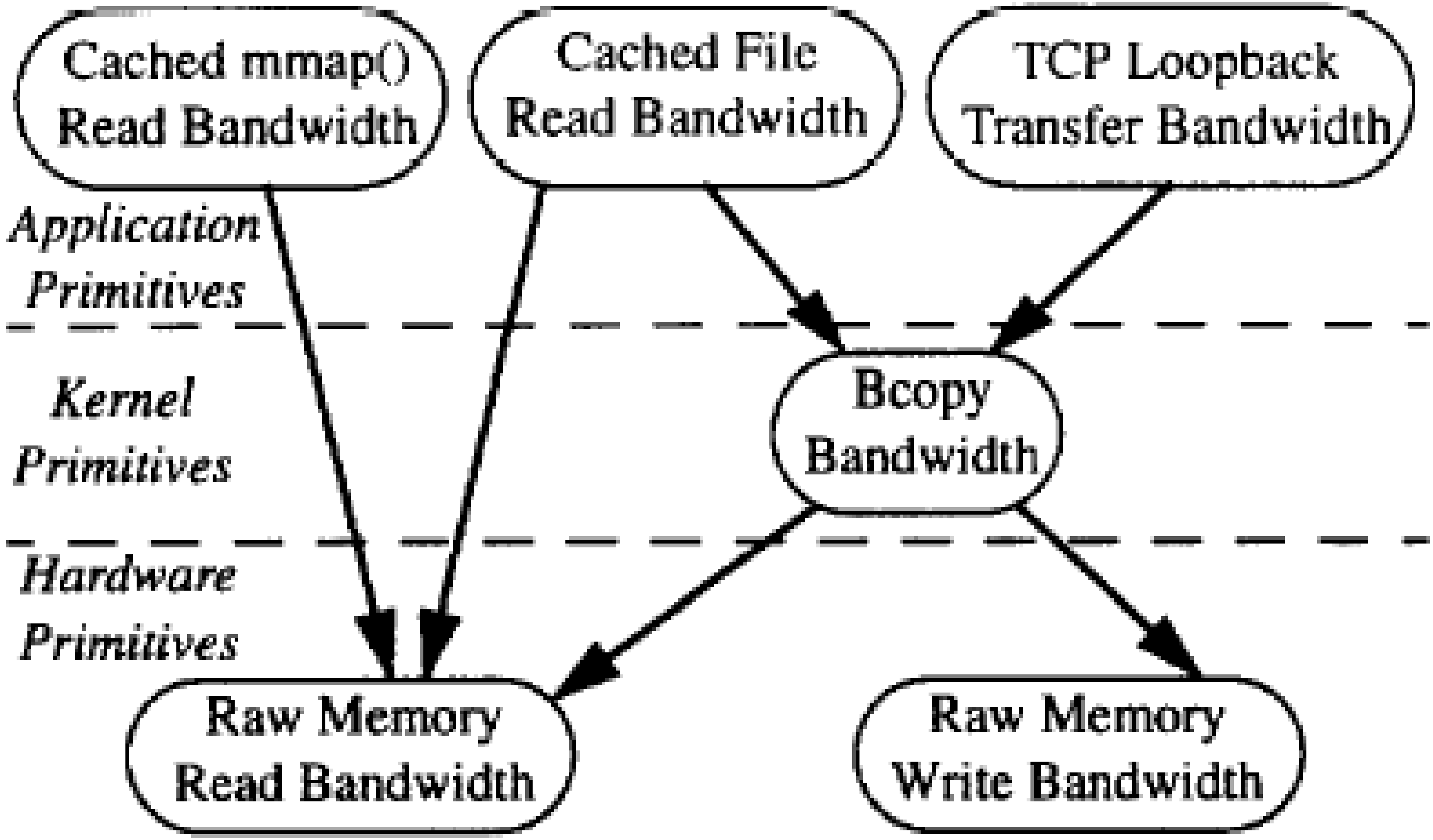
SIGMETRICS 1997

- Systems are built in layers
  - hardware primitives
  - low-level operating system primitives
  - high-level operating system services
  - user applications
- Performance of applications depends on interactions among the lower components
- To understand performance, need to

  1) measure the different primitives in isolation

  2) characterize combinations and interactions

Example: decomposition of bulk data transfer



Raw memory bandwidth

- Dependence on benchmark
  - max BW achieved by walking prearranged pointers
  - more realistic to include indexing of array
- Dependence on hardware features
  - memory technology
  - bus width
  - bus clock rate and its relation to CPU clock rate
  - support for burst transfers on bus (avoid need for bus negotiation)
  - combined writes from cache (writing complete line avoids need to first read and then modify)
- Many delicate details

Kernel service and application bandwidth

- Based on hardware primitive bandwidth we can predict bandwidth at higher levels
  - copy BW = ½ harmonic mean of read BW, write BW
- Deviations indicate interaction with some other aspect of the system
- Example: alternating reads and writes may require different pattern of negotiations for bus

Practical insight:

- Performance depends on intricate details
- Very hard to predict
- Very sensitive to unknown bottlenecks or incompatibilities
- For dedicated-system procurement, better to use application-level benchmarks