10. Performance

Overview

This is not a course about performance analysis or about writing efficient programs, although it often touches on these topics. Both are much too large to be covered, even superficially, in a single lecture devoted to performance. There are many books on performance analysis[1] and a few on efficient programs[2].

Our goal in this handout is more modest: to explain how to take a system apart and understand its performance well enough for most practical purposes. The analysis is necessarily rather rough and ready, but nearly always a rough analysis is adequate, often it's the best you can do, and certainly it's much better than what you usually see, which is no analysis at all. Note that performance *analysis* is not the same as performance *measurement*, which is more common.

What is performance? The critical measures are *bandwidth* and *latency*. We neglect other aspects that are sometimes important: availability (discussed later when we deal with replication), connectivity (discussed later when we deal with switched networks), and storage capacity

When should you work on performance? When it's needed. Time spent speeding up parts of a program that are fast enough is time wasted, at least from any practical point of view. Also, the march of technology, also known as Moore's law, means that in 18 months from March 2006 a computer will cost the same but be twice as fast[3] and have twice as much RAM and four times as much disk storage; in five years it will be ten times as fast and have 100 times as much disk storage. So it doesn't help to make your system twice as fast if it takes two years to do it; it's better to just wait. Of course it still might pay if you get the improvement on new machines as well, or if a 4 x speedup is needed.

How can you get performance? There are techniques for making things faster:
  better algorithms,
  fast paths for common cases, and
  concurrency.
And there is methodology for figuring out where the time is going:
  analyze and measure the system to find the bottlenecks and the critical parameters that determine its performance, and
  keep doing so both as you improve it and when it's in service.
As a rule, a rough back-of-the-envelope analysis is all you need. Putting in a lot of detail will be a lot of work, take a lot of time, and obscure the important points.

[1] Try R. Jain, *The Art of Computer Systems Performance Analysis*, Wiley, 1991, 720 pp.
[2] The best one I know is J. Bentley, *Writing Efficient Programs*, Prentice-Hall, 1982, 170 pp.
[3] A new phenomenon as of 2006 is that the extra speed is likely to come mostly in the form of concurrency, that is, several processors on the chip, rather than a single processor that is twice as fast. This is because the improvements in internal processor architecture that have made it possible to use internal concurrency to speed up a processor that still behaves as though it is executing instructions sequentially are nearly played out.

What is performance: bandwidth and latency

Bandwidth and latency are usually the important metrics. Bandwidth tells you how much work gets done per second (or per year), and latency tells you how long something takes from start to finish: to send a message, process a transaction, or referee a paper. In some contexts it's customary to call these things by different names: throughput and response time, or capacity and delay. The ideas are exactly the same.

Here are some examples of communication bandwidth and latency on a single link. Note that all the numbers are in bytes/sec; it's traditional to quote bandwidths for some interconnects in bits/sec, so be wary of numbers you read.

Medium	*Link*	*Bandwidth*		*Latency*		*Width*
Pentium 4 chip	on-chip bus	30	GB/s	.4	ns	64
PC board	Rambus bus	1.6	GB/s	75	ns	16
	PCI I/O bus	533	MB/s	200	ns	32
Wires	Serial ATA (SATA)	300	MB/s	200	ns	1
	SCSI	40	MB/s	500	ns	32
LAN	Gigabit Ethernet	125	MB/s	100 +	µs	1
	Fast Ethernet	12.5	MB/s	100 +	µs	1
	Ethernet	1.25	MB/s	100 +	µs	1

Here are examples of communication bandwidth and latency through a switch that interconnects multiple links.

Medium	*Switch*	*Bandwidth*		*Latency*		*Links*
Pentium 4 chip	register file	180	GB/s	.4	ns	6
Wires	Cray T3E	122	GB/s	1	µs	2K
LAN	Ethernet switch	4	GB/s	4–100	µs	32
Copper pair	Central office	80	MB/s	125	µs	50K

Finally, here are some examples of other kinds of work, different from simple communication.

Medium	*Bandwidth*		*Latency*	
Disk	40	MB/s	10	ms
RPC on Giganet with VIA	30	calls/ms	30	µs
RPC	3	calls/ms	1	ms
Airline reservation transactions	10000	trans/s	1	sec
Published papers	20	papers/yr	2	years

Specs for performance

How can we put performance into our specs? In other words, how can we specify the amount of real time or other resources that an operation consumes? For resources like disk space that are controlled by the system, it's quite easy. Add a variable `spaceInUse` that records the amount of disk space in use, and to specify that an operation consumes no more than `max` space, write

```
<< VAR used: Space | used <= max => spaceInUse := spaceInUse + used >>
```

This is usually what you want, rather than saying exactly how much space is consumed, which would restrict the code too much.

Doing the same thing for real time is a bit trickier, since we don't usually think of the advance of real time as being under the control of the system. The spec, however, has to put a limit on how much time can pass before an operation is complete. Suppose we have a procedure `P`. We can specify `TimedP` that takes no more than `maxPLatency` to complete as follows. The variable `now` records the current time, and `deadlines` records a set of latest completion times for operations in progress. The thread `Clock` advances `now`, but not past a deadline. An operation like `TimedP` sets a deadline before it starts to run and clears it when it is done.

```
VAR now      : Time
    deadlines: SET Time

THREAD Clock() = DO now < deadlines.min => now + := 1 [] SKIP OD

PROC TimedP() = VAR t : Time
    << now < t /\ t < now + maxPLatency /\ ~ t IN deadlines =>
        deadlines := deadlines + {t} >>;
    P();
    << deadlines := deadlines - {t}; RET >>
```

This may seem like an odd way of doing things, but it does allow exactly the sequences of transitions that we want. The alternative is to construct `P` so that it completes within `maxPLatency`, but there's no straightforward way to do this.

Often we would like to write a probabilistic performance spec; for example, service time is drawn from a normal distribution with given mean and variance. There's no way to do this directly in Spec, because the underlying model of non-deterministic state machines has no notion of probability. What we can do is to keep track of actual service times and declare a failure if they get too far from the desired form. Then you can interpret the spec to say: either the observed performance is a reasonably likely consequence of the desired distribution, or the system is malfunctioning.

How to get performance: Methodology

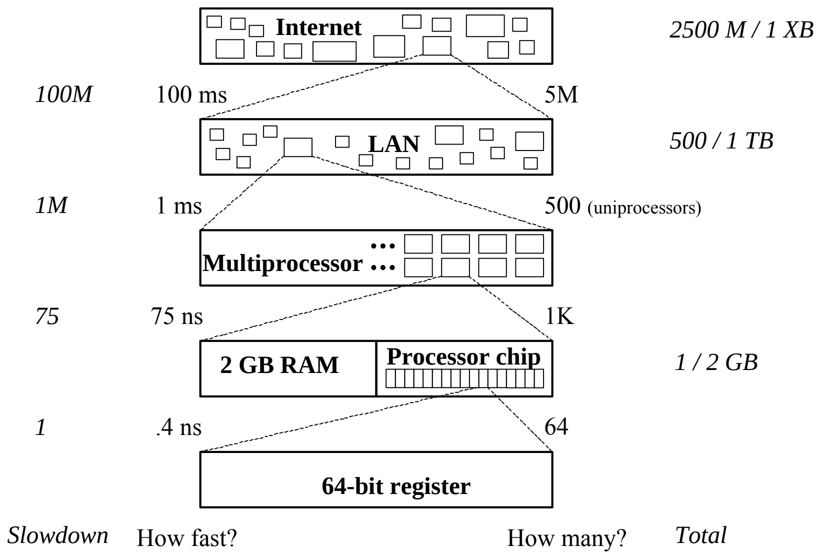
First you have to choose the right scale for looking at the system. Then you have to model or analyze the system, breaking it down into a few parts that add up to the whole, and measure the performance of the parts.

Choosing the scale

The first step in understanding the performance of a system is to find the right scale on which to analyze it. The figure shows the scales from the processor clock to an Internet access; there is a range of at least 50 million in speed and 50 million in quantity. Usually there is a scale that is the right one for understanding what's going on. For the performance of an inner loop it might be the system clock, for a simple transaction system the number of disk references, and for a Web browser the number of IP packets.

In practice, systems are not deterministic. Even if there isn't inherent non-determinism caused by unsynchronized clocks, the system is usually too complex to analyze in complete detail. The way to simplify it is to approximate. First find the right scale and the right primitives to count, ignoring all the fine detail. Then find the critical parameters that govern performance at that scale: number of RPC's per transaction, cache miss rate, clock ticks per instruction, or whatever.

In this way you should be able to find a simple formula that comes within 20% of the observed performance, and usually this is plenty good enough.



Scales of interconnection. Relative speed and size are in italics.

For example, in the 1994 election DEC ran a Web server that provided data on the California election. It got about 80k hits/hour, or 20/sec, and it ran on a 200 MIPS machine. The data was probably all in memory, so there were no disk references. A hit typically returns about 2 KB of data. So the cost was about 10M instructions/hit, or 5K instructions/byte returned. Clearly this was not an optimized system.

By comparison, a simple debit-credit transaction (the TPC-A benchmark) when carefully coded does slightly more than two disk i/o's per transaction (these are to read and write per-account data that won't fit in memory). If carefully coded it takes about 100K instructions. So on a 2000 MIPS machine it will consume 50 µs of compute time. Since two disk i/o's is 20 ms, it takes 400 disks to keep up with this CPU for this application. Since this is not too reasonable, engineers have responded by coding transactions less carefully, taking advantage of the fact that instructions are so cheap.

As a third example, consider sorting 10 million 64 bit numbers; the numbers start on disk and must end up there, but you have room for the whole 80 MB in memory. So there's 160 MB of disk transfer plus the in-memory sort time, which is *n* log *n* comparisons and about half that many swaps. A single comparison and half swap might take 10 instructions with a good code for Quicksort, so this is a total of 10 * 10 M * 24 = 2.4 G instructions. Suppose the disk system can transfer 80 MB/sec and the processor runs at 200 MIPS. Then the total time is 2 sec for the disk plus 1.2 sec for the computing, or 3.2 sec, less any overlap you can get between the two phases.

With considerable care this performance can be achieved. On a parallel machine you can do perhaps 30 times better.[4]

Here are some examples of parameters that might determine the performance of a system to first order: cache hit rate, fragmentation, block size, message overhead, message latency, peak message bandwidth, working set size, ratio of disk reference time to message time.

Modeling

Once you have chosen the right scale, you have to break down the work at that scale into its component parts. The reason this is useful is the following principle:

> If a task *x* has parts *a* and *b*, the cost of *x* is the cost of *a* plus the cost of *b*, plus a system effect (caused by contention for resources) which is usually small.

Most people who have been to school in the last 20 years seem not to believe this. They think the 'system effect' is so large that knowing the cost of *a* and *b* doesn't help at all in understanding the cost of *x*. But they are wrong. Your goal should be to break down the work into a small number of parts, between two and ten. Adding up the cost of the parts should give a result within 10% of the measured cost for the whole.

If it doesn't then either you got the parts wrong (very likely), or there actually *is* an important system effect. This is not common, but it does happen. Such effects are always caused by *contention* for resources, but this takes two rather different forms:

- *Thrashing* in a cache, because the sum of the working sets of the parts exceeds the size of the cache. The important parameter is the cache miss rate. If this is large, then the cache miss time and the working set are the things to look at. For example, SQL server on Windows NT running on a DEC Alpha 21164 in 1997 executes .25 instructions/cycle, even though the processor chip is capable of 2 instructions/cycle. The reason turns out to be that the instruction working set is much larger than the instruction cache, so that essentially every block of 4 instructions (16 bytes or one cache line) causes a cache miss, and the miss takes 64 ns, which is 16 4 ns cycles, or 4 cycles/instruction.
- *Clashing* or queuing for a resource that serves one customer at a time (unlike a cache, which can take away the resource before the customer is done). The important parameter is the queue length. It's important to realize that a resource need not be a physical object like a CPU, a memory block, a disk drive, or a printer. Any lock in the system is a resource on which queuing can occur. Typically the physical resources are instrumented so that it's fairly easy to find the contention, but this is often not true for locks. In the Alta Vista web search engine, for example, CPU and disk utilization were fairly low but the system was saturated. It turned out that queries were acquiring a lock and then page faulting; during the page fault time lots of other queries would pile up waiting for the lock and unable to make progress.

In the section on techniques we discuss how to analyze both of these situations.

[4] Andrea Arpaci-Dusseau et al., High-performance sorting on networks of workstations. SigMod 97, Tucson, Arizona, May, 1999, http://now.cs.berkeley.edu/NowSort/nowSort.ps .

Measuring

The basic strategy for measuring is to count the number of times things happen and observe how long they take. This can be done by sampling (what most profiling tools do) or by logging significant events such as procedure entries and exits. Once you have collected the data, you can use statistics or graphs to present it, or you can formulate a model of how it should be (for example, time in this procedure is a linear function of the first parameter) and look for disagreements between the model and reality.[5] The latter technique is especially valuable for continuous monitoring of a running system. Without it, when a system starts performing badly in service it's very difficult to find out why.

Measurement is usually not useful without a model, because you don't know what to do with the data. Sometimes an appropriate model just jumps out at you when you look at raw profile data, but usually you have to think about it and try a few things. This is just like any branch of science: without a theory you can't make sense of the data.

How to get performance: Techniques

There are three main ways to make your program run faster: use a better algorithm, find a common case that can be made to run fast, or use concurrency to work on several things at once.

Algorithms

There are two interesting things about an algorithm: the 'complexity' and the 'constant factor'. An algorithm that works on n inputs can take roughly k (constant) time, or $k \log n$ (logarithmic), or $k\, n$ (linear), or $k\, n^2$ (quadratic), or $k\, 2^n$ (exponential). The k is the constant factor, and the function of n is the complexity. Usually these are 'asymptotic' results, which means that their percentage error gets smaller as n gets bigger. Often a mathematical analysis gives a worst-case complexity; if what you care about is the average case, beware. Sometimes a 'randomized' algorithm that flips coins internally can make the average case overwhelmingly likely.

For practical purposes the difference between $k \log n$ time and constant time is not too important, since the range over which n varies is likely to be 10 to 1M, so that $\log n$ varies only from 3 to 20. This factor of 6 may be much less than the change in k when you change algorithms. Similarly, the difference between $k\, n$ and $k\, n \log n$ is usually not important. But the differences between constant and linear, between linear and quadratic, and between quadratic and exponential are very important. To sort a million numbers, for example, a quadratic insertion sort takes a trillion operations, while the $n \log n$ Quicksort takes only 20 million in the average case (unfortunately the worst case for Quicksort is also quadratic). On the other hand, if n is only 100, then the difference among the various complexities (except exponential) may be less important than the values of k.

Another striking example of the value of a better algorithm is 'multi-grid' methods for solving the n-body problem: lots of particles (atoms, molecules or asteroids) interacting according to some force law (electrostatics or gravity). By aggregating distant particles into a single virtual particle, these methods reduce the complexity from n^2 to $n \log n$, so that it is feasible to solve systems with millions of particles. This makes it practical to compute the behavior of complex chemical reactions, of currents flowing in an integrated circuit package, or of the solar system.

[5] See Perl and Weihl, Performance assertion checking. *Proc. 14th ACM Symposium on Operating Systems Principles*, Dec. 1993, pp 134-145.

Fast path

If you can find a common case, you can try to do it fast. Here are some examples.

Caching is the most important: memory, disk (virtual memory, database buffer pool), web cache, memo functions (also called 'dynamic programming'), ...

Receiving a message that is an expected ack or the next message in sequence.

Acquiring a lock when no one else holds it.

Normal arithmetic vs. overflow.

Inserting a node in a tree at a leaf, vs. splitting a node or rebalancing the tree.

Here is the basic analysis for a fast path.

1 = fast time, 1 << 1 + *s* = slow time, *m* = miss rate = probability of taking the slow path.

```
t = time = 1 + m * s
```

There are two ways to look at it:

The slowdown from the fast case (time 1). If *m* = 1/*s* then *t* = 2, a 2 x slowdown.

The speedup from the slow case (time *s*). If *m* = 50% then *t* = *s*/2 + 1, nearly a 2 x speedup,

You can see that it makes a big difference. For *s* = 100, a miss rate of 1% yields a 2 x slowdown, but a miss rate of 50% yields a 2 x speedup.

The analysis of fast paths is most highly developed in the study of computer architecture.[6]

Batching has the same structure:

1 = unit cost, *s* = startup (per-batch) cost, *b* = batch size.

```
t = time = (b + s) / b = 1 + s/b
```

So *b* is like 1/*m*. Amdahl's law for concurrency (discussed below) also has the same structure.

Concurrency with lots of small jobs

Usually concurrency is used to increase bandwidth. It is easiest when there are lots and lots 'independent' requests, such as web queries or airline reservation transactions. Some examples: customers buying books at Amazon, web searches on Google, or DNS name lookups. In this kind of problem there is no trouble getting enough concurrent work to do, and the challenge is getting it done efficiently. The main obstacle is bottlenecks.

Getting more and more concurrency to work is called *scaling* a system. Scaling is increasingly important because the internet allows demand for a popular service to grow almost without bound, and because commodity components are so much cheaper per unit of raw performance than any other kind of component. For a system to scale:

- It must not have any algorithms with complexity worse than log *n*, where *n* is the number of components.

[6] Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1995. The second edition has a great deal of new material.

- It must not have bottlenecks.

Both of these are easy when there is no shared data, and get harder as you increase the amount of shared data, the rate at which it changes, and the requirement for consistency.

For example, in the Domain Name System (DNS) that maps a name like `lcs.mit.edu` into an IP address, the root is potentially a bottleneck, since in principle every lookup must start at the root. If the root were a single machine, DNS would be in big trouble. Two techniques relax this bottleneck in the real DNS:

Caching the result of lookups in the root, on the theory that the IP address for `mit.edu` changes very seldom. The price, of course, is that when it does change there is a delay before everyone notices.

Many copies of the root that are loosely consistent, on the theory that when a name is added or changed, it's not essential for all the copies to find out about it atomically.

Returning to concurrency for bandwidth, there can be multiple identical resources or several distinct resources. In the former case the main issue is load balancing (there is also the cost of switching a task to a particular resource). The most common example is multiple disks. If the load if perfectly balanced, the i/o rate from *n* disks is *n* times the rate from one disk. The debit-credit example above showed how this can be important. Getting the load perfectly balanced is hard; in practice it usually requires measuring the system and moving work among the resources. It's best to do this automatically, since the human cost of doing it manually is likely to dwarf the savings in hardware.

When the resources are distinct, we have a 'queuing network' in which jobs 'visit' different resources in some sequence, or in various sequences with different probabilities. Things get complicated very quickly, but the most important case is quite simple. Suppose there is a single resource and tasks arrive independently of each other ('Poisson arrivals'). If the resource can handle a single request in a service time *s*, and its utilization (the fraction of time it is busy) is *u*, then the average request gets handled in a response time

```
r = s/(1 - u)
```

The reason is that a new request needs to get s amount of service before it's done, but the resource is only free for 1 - *u* of the time. For example, if *u* = .9, only 10% of the time is free, so it takes 10 seconds of real time to accumulate 1 second of free time.

Look at the slowdowns for different utilizations.

2 x at 50%
10 x at 90%
Infinite at 100% ('saturation')

Note that this rule applies only for Poisson (memoryless or 'random' arrivals). At the opposite extreme, if you have periodic arrivals and the period is synchronized with the service time, then you can do pipelining, drop each request into a service slot that arrives soon after the request, and get *r* = *s* with *u* = 1. One name for this is "systolic computing".

A high *u* has two costs: increased *r*, as we saw above, and increased sensitivity to changing load. Doubling the load when *u* = .2 only slows things down by 30%; doubling from *u* = .8 is a catastrophe. High *u* is OK if you can tolerate increased *r* and you know the load. The latter could be because of predictability, for example, a perfectly scheduled pipeline. It could also be because of aggregation and statistics: there are enough random requests that the total load varies very little. Unfortunately, many loads are "bursty", which means that requests are more likely to follow other requests; this makes aggregation less effective.

When there are multiple requests, usually one is the bottleneck, the most heavily loaded component, and you only have to look at that one (of course, if you make it better then something else might become the bottleneck).

Servers with finite load

Many papers on queuing theory analyze a different situation, in which there is a fixed number of customers that alternate between thinking (for time z) and waiting for service (for the response time z). Suppose the system in steady state (also called 'equilibrium' or 'flow balance'), that is, the number of customers that demand service equals the number served, so that customers don't pile up in the server or drain out of it. You can find out a lot about the system by just counting the number of customers that pass by various points in the system

A customer is in the server if it has entered the server (requested service) and not yet come out (received all its service). If there are n customers in the server on the average and the throughput (customers served per second) is x, then the average time to serve a customer (the response time) must be $r = n/x$. This is "Little's law", usually written $n = rx$. It is obvious if the customers come out in the same order they come in, but true in any case. Here n is called the "queue length", though it includes the time the server is actually working as well.

If there are N customers altogether and each one is in a loop, thinking for z seconds before it enters the server, and the throughput is x as before, then we can use the same argument to compute the total time around the loop $r + z = N/x$. Solving for r we get $r = N/x - z$. This formula doesn't say anything about the service time s or the utilization u, but we also know that the throughput $x = u/s$ ($1/s$ degraded by the utilization). Plugging this into the equation for r we get $r = Ns/u - z$, which is quite different from the equation $r = s/(1 - u)$ that we had for the case of uniform arrivals. The reason for the difference is that the population is finite and hence the maximum number of customers that can be in the server is N.

Concurrency in a single job

In using concurrency on a single job, the goal is to reduce latency—the time to get the job done. This requires a parallel algorithm, and runs into Amdahl's law, which is another kind of fast path analysis. In this case the fast path is the part of the program that can run in parallel, and the slow path is the part that runs serially. The conclusion is the same: if you have 100 processors, then your program can run 100 times faster if it all runs in parallel, but if 1% of it runs serially then it can only run 50 times faster, and if half runs serially then it can only run twice as fast. Usually we take the slowdown view, because the ideal is that we are paying for all the processors and so every one should be fully utilized. Then a 99% parallel / 1% serial program, which achieves a speedup of 50, is only half as fast as our ideal. You can see that it will be difficult to make efficient use of 100 processors on a single job.

Another way of looking at concurrency in a single job is the following law (actually a form of Little's law, discussed above from a different point of view):

$$\boxed{\textit{concurrency} = \textit{latency} \times \textit{bandwidth}}$$

As with Ohm's law, the way you look at this equation depends on what you think are the independent variables. In a CPU/memory system, for example, the latency of a cache miss is

fixed at about 100 ns. Suppose the CPU has a floating-point unit that can do 3 multiply-add operations per ns (typical for 2006). In a large job each such operation will require one operand from main memory because all the data won't fit into the cache; multiplying two large matrices is a simple example. So the required memory bandwidth to keep the floating point unit busy is 300 reads/100 ns. With latency and bandwidth fixed, the *required* concurrency is 300.

On the other hand, the *available* concurrency is determined by the program and its execution engine. For this example, you must have 300 outstanding memory reads (nearly) all the time. A read is outstanding if it has been issued by the CPU, but the data has not yet been consumed. This could be accomplished by having 300 threads, each with one outstanding read, or 30 threads each with 10 reads, or 1 thread with 300 reads.

How can you have a read outstanding? In a modern CPU with out-of-order execution and register renaming, this means that the FETCH operation has been issued and a register assigned to hold the result. Before the result returns, the CPU can keep going, issue an ADD that uses the memory result, assign the result register for the ADD, and continue issuing instructions. If there's a branch on the result, however, the CPU must guess the branch result, since proceeding down both paths quickly becomes too expensive; the polite name for this is 'branch prediction'. It can continue down the predicted path 'speculatively', which means that it has to back up if the guess turns out to be wrong. On typical jobs of this kind prediction errors are < 1%, so speculative execution can continue for quite a while. More registers are needed, however, to hold the data needed for backing up. Eventually either the CPU runs out of registers, or the chances of a bad prediction are large enough that it doesn't pay to keep speculating.

You can see that it's hard to keep all these balls in the air long enough to get 300 reads outstanding nearly all the time, and no existing CPU does so. Some CPUs get hundreds of outstanding reads by using 'vector' instructions that call for many reads or adds in a single instruction, for example, "read 50 values from addresses a, a+100, a+200, ... into vector register 7" or "add vector register 7 to vector register 9 and put the result in vector register 13". Such instructions are much less flexible than scalar reads, but they can use many fewer CPU resources to make a read outstanding. Somewhat more flexible operations like "read from the addresses in vector register 7 and put the results into vector register 9" are possible; they help with sparse matrices.

Multiple threads reduce the coupling among outstanding operations. In fact, with 300 threads, each one would need only one outstanding operation, so there would be no need for speculative execution and backup. Scheduling must be done by the hardware, however, since you have to switch threads after every memory operation. Keeping so many threads running requires lots of hardware resources. In fact, it requires many of the same hardware resources required for a single thread with lots of outstanding operations. High-volume CPUs currently can run 2 threads at a time, so we are some distance from the goal.

This formulation of Little's law is useful for understanding not just CPU/memory systems, but any system in which you are trying to get high bandwidth with a fixed latency. It tells you how much concurrency you need. Then you must ask whether there's a source for that much concurrency, and whether there are enough resources to maintain the internal state that it requires. In the 'embarrassingly parallel' applications of the previous section there are plenty of requests, and a job that's waiting just consumes some memory, which is usually in ample supply. This means that you only have to worry about the costs of the data structures for scheduling.

Summary

Here are the most important points about performance.

- Moore's law: The performance of computer systems at constant cost doubles every 18 months, or increases by ten times every five years.
- To understand what a system is doing, first do a back-of-the-envelope calculation that takes account only of the most important one or two things, and then measure the system. The hard part is figuring out what the most important things are.
- If a task *x* has parts *a* and *b*, the cost of *x* is the cost of *a* plus the cost of *b*, plus a system effect (caused by contention for resources) which is usually small.
- For a system to scale, its algorithms must have complexity no worse than log *n*, and it must have no bottlenecks. More shared data makes this harder, unless it's read-only.
- The time for a task which has a fast path and a slow path is $1 + m * s$, where the fast path takes time 1, the slow path takes time $1 + s$, and the probability of taking the slow path is *m* (the miss rate). This formula works for batching as well, where the batch size is $1/m$.
- If a shared resource has service time *s* to serve one request and utilization *u*, and requests arrive independently of each other, then the response time is $s/(1 - u)$. It tends to infinity as *u* approaches 1.
- concurrency = latency × bandwidth

.