Chapter 1

Introduction 

Computers are synonymous with speed. Early computers were expensive and unreliable, but, compared to the alternatives, they were *fast*. Whether analyzing census data, decrypting enemy messages, calculating payroll — any of dozens of important tasks – the machines left humans in the dust. But not by enough. Decades have passed, yet the demand for greater speed remains unsated, and making systems faster remains an obsession for computer scientists and engineers. They quest relentlessly for faster processors, faster memory, faster peripherals, faster algorithms, faster data structures. Users, implementers, and researchers have struck a single tone for a half century: fast is good, and faster is better.

There are good reasons for this. For one, *speed is engaging*. Users thrive on snappy, responsive interactions. They may tolerate slow systems, but they *enjoy* fast ones. Second, *speed is enabling*. Throw enough speed at a problem, and it emerges from research labs and becomes part of everyday life. Take hours to generate a video frame, and you get a feature film every year or two. Generate 30 or more frames per second, and you've hooked a sizable portion of humanity on video games. Third, *speed means money*. Engines, factories, businesses, and financial markets are all more efficient when information is disseminated quickly. Productivity goes up when the time taken to perform a task goes down. Time is money, the saying goes, and the saying's not kidding.

Speed has always been one of the most important characteristics of software systems. Curiously, the pursuit of *fastware* – fast software systems – has acquired a sullied reputation. It's not uncommon for programmers thinking about how to increase the speed of their systems to face colleagues who are dismissive, skeptical, or even hostile.

Bring up the topic of designing or implementing for speed, and you'll often be confronted with *Speed Cliché #1*:

First make it right, then make it fast.

This is fundamentally misguided, because it implies that "right" and "fast" are independent. They're not. Every system has a point where it's so slow, it's not acceptable. Even if no speed requirement is present in the system's specification, acceptable speed is mandatory, and that means that s*peed is a correctness criterion*. The system simply can't be right unless it's fast enough. The annals of soft-

ware ignominy are littered with "correct" systems that were too slow to be useful. People who play "correct" and "fast" off against one another fail to grasp that "fast" is a crucial part of "correct."

A second problem with the "right first, fast second" way of thinking is that it assumes that a slow system can be transformed into a behaviorally equivalent fast one with relatively little and largely localized effort. This is often not the case. If the way to make a slow system fast is to add parallelism, but the system's algorithms and data structures were designed only for serial access, there's rarely a simple, localized way to adapt them to fast parallel use. If the slow system was designed to solve a general problem, and the way to speed it up is to take advantage of the restricted cases that will typically be encountered, making it fast may entail an entirely new design and implementation. "Fast" isn't a fashion accessory you sew onto a system after it's done. It's an integral part of the system's fabric.

Should you be fortunate enough to avoid Speed cliché #1, or should you encounter it and break its rasping hold, you're likely to confront *Speed Cliché #2*:

Premature optimization is the root of all evil.

This cliché commands respect, because it was written by Donald Knuth, the man whose multi-volume monograph, *The Art of Computer Programming*,† all but codified computer science. From compiler development to algorithm analysis to typesetting and more, Knuth's accomplishments are legendary. You don't mess with Knuth. If it's you versus Knuth, you're going to lose. And Knuth's wrote what has become Speed cliché #2.

That cliché, it turns out, is from a paper discussing low-level goto-related code optimizations,‡ part of the fallout from Dijkstra's famous "goto considered harmful" letter.* The particular code transformation under discussion is the rewrite of a short inner loop to reduce the number of tests per iteration from two to one. Regarding such low-level code transformations, Knuth writes:

> It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail.

Knuth's argument is that low-level code tweaks aren't justified without empirical data showing that the code in question has a noticeable effect on overall performance. He doesn't argue against taking performance concerns into account dur-

† Published by Addison-Wesley in various editions and years from 1968 to the present.

‡ Knuth, Donald E., "Structured Programming with go to Statements," *Computing Surveys*, December 1974.

* Dijkstra, E. W. "Go to statement considered harmful," *Communications of the ACM*, March 1968, pg. 147-148. Notes Knuth in his citation: "There are two instances of pages 147-148 in this volume; the second 147-148 is relevant here."

ing requirements specification, architecture or design, or selection of algorithms and data structures. If you're thinking about the speed aspects of your system during any of these phases of development, Speed Cliché #2 doesn't apply. If you're looking for ways to make your code faster via low-level modifications, and you've got empirical data backing your belief that the code you're looking at affects the overall speed of your system, Speed Cliché #2 doesn't apply. In fact, if you're thinking of writing your code a particular way, because experience has shown that that way generates better code without unduly compromising maintainability, Knuth makes clear that he's on your side:

> The conventional wisdom shared by many of today's software engineers calls for ignoring efficiency in the small; but I believe this is simply an overreaction to the abuses they see being practiced by pennywise-and-pound-foolish programmers, who can't debug or maintain their "optimized" programs. In established engineering disciplines a 12% improvement [the amount under discussion at that point in the paper], easily obtained, is never considered marginal; and I believe the same viewpoint should prevail in software engineering.

Speed Cliché #2 really stops you only if you're spending more than negligible amounts of time tinkering with low-level code in the absence of data suggesting that the tinkering will make the system faster. And if that's what you're doing, you deserve to be stopped.

Speed aficionados who've endured the cliché competition sometimes run into a new objection: that modern computers are so fast, programmers no longer need worry about the speed of their systems – that doing so is a waste of valuable developer time. Situations where this is true surely exist, but the evidence suggests that they may be the exception rather than the rule. Application areas where speed is a significant concern are not just common, they're ubiquitous. For example:

- **High-performance servers**. Regardless of whether you're serving requests for web pages, streaming media, geographical data, DNS lookups, general-purpose database queries, or whatnot, speed matters. Faster is better in almost every way, including latency, throughput, and cost per transaction. High-performance servers are representative of a broad class of software systems that, practically speaking, can *never be fast enough*.
- **Systems software** such as operating systems, virtual machines, hypervisors, database management systems, drivers, programming language libraries – any kind of software used by speed-sensitive systems. Applications of all kinds run on or otherwise use systems software, so if systems software isn't fast, the applications using it can't be, either. If you're working on code called, linked with, or otherwise used by other code, it's a good bet your clients expect your code to sizzle. Systems software also falls into the category of software that can

never be fast enough, because making it faster enables the creation of new higher-level applications that would otherwise run unacceptably slowly.

- **Embedded systems**. Embedded systems are typically designed to perform one or a small number of specific tasks, e.g., real-time monitoring of automotive or industrial processes, implementing network protocols, organizing and playing digital media for consumer devices, etc. Developers of such systems often care about speed not just for its own sake, but because they can trade "excess" speed for other important system characteristics. Software that runs, say, 50% faster than strictly necessary on given hardware makes possible the replacement of that hardware with other components that cost less, draw less power, or exhibit greater reliability. Because there is always something for which "excess" speed can be traded, embedded systems are another example of software than can never be fast enough.
- **Video games**, too, fall into this category, because developers of video games want to fully exploit the hardware available to them. If the CPU(s) aren't pegged at 100% during game play, something's wrong. Programmers for different parts of a game receive a CPU budget (a limit on the number of CPU cycles they're permitted to use per frame), a budget that's generally far less than what's needed to do everything the designers can envision. Free up precious CPU cycles by making one part of the game faster, and the newly-available cycles can be employed by other developers to make the game even better: more aurally and visually compelling, more behaviorally nuanced, and more immersive.
- **Simulations**. The world we live in is but one of many we can imagine. Every day, significant effort goes into modeling alternate realities and simulating their behavior. Firms in the financial industry do it to identify investment opportunities and mitigate risks. Pharmaceutical companies do it to evaluate the effect of prospective drugs. Electronics companies do it to verify chip designs before committing to manufacturing. Governments do it to avoid having to test new designs for nuclear weapons. Almost anything you can think of is being simulated by somebody, and whoever's doing the simulating is using software that can't possibly run fast enough. If only it ran faster! Then they'd be able to simulate more alternate worlds for longer periods of simulated time with greater modeling resolution
- **Real-time financial trading** (e.g., algorithmic trading) is a field where time is literally money. Every time an investment instrument is bought or sold, every time an economic report is issued, every time anything happens that causes traders to react, money is made and lost. Get the right information, analyze it correctly, and act on it while it's still valid, and you're on the road to riches. Your competition knows about this road, too, of course, and if they react to events before you do, their actions can invalidate your analysis. To have an

edge, you have to be faster than they are, and, just like you, they're coming up with clever ways to be faster all the time. It's a speed-based arms race with no end, and the weapons are made of software.

- **Number-crunching applications** such as encryption/decryption, data compression/decompression, image processing, the search for new prime numbers, etc. These applications are also generally never fast enough, because once they run with reasonable speed, we throw bigger problems at them. We use more demanding encryption schemes with larger keys, for example, and we shift from wanting to compress "normal" resolution video to high-definition video. No matter how fast we crunch the numbers, it's never fast enough.
- **Analysis of large data sets**, including data mining applications, Internet search engines, CAD/CAE (computer-aided design/engineering) software, and pretty much anything demanding 64 (or more) bits of address space. Many applications involving such quantities of data currently take hours or days to run, and even then the results may be incomplete or offer resolution that is less than ideal. Because the size of these data sets are always on the rise, and because it's usually the case that the sooner the analyses are completed, the more useful the results, this is yet another area where the software can never run fast enough.

There are, of course, overlaps in the above application areas. Video game and real-time financial trading software typically involves significant simulation, simulation often requires substantial number-crunching on enormous data sets, and big data sets are generally delivered by high-performance servers. Still, the list above should make clear that software systems where speed is not just important, but *really* important, underlie almost everything we do with computers.

Which sets the stage for the final speed-dependent application area I'll mention, the one that probably has the most direct effect on the most people:

- **Interactive applications**. It goes without saying that users dislike delays between when they type something and when it appears, jerky scrolling, pauses while windows repaint, and the many other contradictions to the fluid desktop experience computers are supposed to provide. Research has established that users notice delays of over $1/10^{th}$ second between when they request an action and they receive feedback that the action has been performed. It has been demonstrated that hyperlinks should be traversed in no more than $1/4^{th}$ second. It has also been shown that when the delay between user command and machine response exceeds 1 second, people no longer feel like the experience is truly interactive. Given the long boot and start-up times for many systems and applications, the delays often accompanying reading and writing files, and the prevalence of wait cursors, progress meters, and other harbingers of slowness that commonly plague the end-user experience, it's fair to say that many allegedly interactive systems aren't. If your interactive application isn't routinely offer-

ing a response time of 1/10th second or less (regardless of what it's being asked to do), it needs to be faster.

Many interactive applications are little more than user interfaces atop one or more of the categories of speed-sensitive applications we've already discussed. Web browsers, for example, communicate with high-performance servers, perform number-crunching when decompressing images and engaging in secure communications, and depend on libraries (systems software) for platform-dependent GUI functionality. Given the speed constraints interactive applications are expected to satisfy, this puts additional pressure on the software underneath them to be as fast as possible.

Some people consider speed-sensitive software a boutique item, one of interest primarily to scientists and engineers investigating fluid dynamics and global weather patterns with large clusters of multicomputers. The term *high performance computing* (HPC) has become synonymous with such exceedingly demanding applications. By now, I hope it's clear that the importance of fast software is much more widespread than this, that in fact *most* nontrivial software should be fastware. This book is about how to create it.

The Meaning of "Fast"

Before we can talk about fastware, we have to talk about what it means to be fast. We're concerned with the speed of "actions," where "action" is a general term for an operation that can be requested. For example, using your mouse to click on a link in a web browser is a request to load the page that link refers to, and in that case, we'd be interested in how long it takes to load the page. (Actually, we'd probably be more interested in how long it *appears* to take to load the page, but more on that in a moment.) Making a call to a function† in a library API would constitute a request to do whatever that function is supposed to do, and we'd be concerned with how long it takes (or, again, *appears* to take) to do it. If the action of interest is running a program from a command line or script, our interest would be in how long it takes that program to run (possibly including any time required to get from the calling context into the program and back again to the calling context).

Performance of an action is "fast" if the action *appears to produce an acceptable result quickly*. This definition requires that we look more closely at appearance, at acceptability, and at quickness.

Appearance is important, because what an action does and what it appears to do may not be the same. Consider a function that is specified to return in a buffer all the documents on a network containing a specified search term. The straight-

† I use the term "function" to mean any invokable entity in a program, so my use of the term also covers methods, subroutines, properties – essentially anything you can call.

forward approach is to find all the documents, put them in a buffer, then return. An alternative approach is to create an empty buffer, spawn multiple threads or processes to concurrently do the searching and placement of results in the buffer, then return to the caller a reference to the buffer. With this approach, the job *appears* to be finished when the function returns, but the spawned threads or processes may actually add results to the buffer for some time.

As another example, suppose a function is specified to draw a building to be used as part of the scenery in a video game. The obvious way to do this is to take a 3D model of the building, then render it. An alternative is to draw a 2D outline of the building, then apply a texture map so that it looks like a building. For many video games purposes, it *appears* to have produced the same result as the more resource-intensive rendering of the 3D model.

Whether an action appears to do what it's supposed to do is determined by how the client (the person or code requesting the action) is permitted to investigate the result. The more closely the permitted examination, the more difficult it is to do something different (for speed purposes, typically *less*) from what was promised. Under no conditions may an action contradict its specification; if it does, it's simply wrong. But a well-chosen specification, one tuned for the expected uses of the action-invocation interface (i.e., user interface or API), can allow for implementations that appear to do what they are supposed to, but actually do something somewhat different.

What they do must be **acceptable**, of course, because otherwise the result is useless, regardless of how fast it's accomplished. Sometimes an action that's not really correct is still acceptable. For example, suppose you search for "giant anteater" using an Internet search engine. Informally, you're requesting the most relevant documents on the Internet matching this query, but no search engine will even try to find them for you. For one thing, they're not going to search the Internet. Instead, they'll consult the data their web-crawlers have gathered about documents they've seen on the Internet, and they'll compose their query result from that. Since they haven't seen all pages on the Internet, and since the ones they have seen were examined at some point in the past, the best they'll be able to do is give you a list of the best hits for the part of the Internet they've searched based on what was in those documents at the time they indexed them. But you may not get even that. The humongabytes of data the search engine has collected† is spread over an army of servers, and at the time you make your request, some of those servers may fail or be too busy to reply in a timely fashion. If so, the contributions they'd have made to the result of your request will simply be omitted. Conceptually, you asked for the most relevant *current* documents on the *entire* Internet, and what you got back was the most relevant documents *last seen* on the *part* of the Internet they'd looked at and *that was fetched before the query timed*

† Probably petabytes as I write this, possibly exebytes or zettabytes by the time you read it.

out. Hardly the same. But you rarely care, because the result is still acceptable (and the alternative – giving you what you really asked for – is, in this case, essentially impossible to provide).

Which brings us to **quickly**. Saying that an action is fast if it is performed quickly seems like nothing more than replacing a poorly-defined adjective with an equally poorly-defined adverb, but whether something is fast is, in the end, subjective. Still, we can make some useful observations. First, if the action is requested by a human and the action is accomplished in no more than about $1/10^{th}$ of a second, the human perceives the action as instantaneous, and humans can't detect differences in speed among instantaneous actions. There is thus a limit on how fast things can appear to humans. There's also a perception limit when the entity requesting an action is software, but software's ability to resolve tiny differences in time, combined with the fact that technology typically gets better with each generation, means that there is no practical limit to the ability of software to detect that one thing is faster than another.

It is at this point that expectations enter the picture. Superficially, something is fast if somebody thinks it is. (There's always a "somebody" involved. Every API call is ultimately made on behalf of some program being run, directly or indirectly, by and for some person.) As a general rule, "fast" means "faster than expected," and therein lies the problem. First, "fast" doesn't mean "as fast as expected," because "as fast as expected" isn't fast, it's "normal." So if you expect an operation to take 100 milliseconds and it takes 80, that's fast. But not for long. After a while, 80 milliseconds becomes the new normal. 100 – the old normal – is relegated to "slow," and "fast" now means less than 80 milliseconds. In the realm of automobiles, there's a name for this "fast becomes normal as soon as I get used to it" phenomenon: *velocitization*. I don't know of a name for the same thing in software, but examples of it abound. Remember the blazingly fast 56K modems we used to marvel at? Are you still marveling? How about that speedy computer you bought a few years ago? Does it still seem speedy? There's an important lesson here. People's evaluation of what's "fast" changes over time, becoming always more demanding until, for interactive applications, the action in question is perceived to occur instantaneously. For noninteractive applications, including the many categories of software that can never run fast enough, the need to continually increase speed in order to retain the "fast" label is never-ending. Because "normal" speed is what "fast" used to be not long ago, writing software whose speed will satisfy user expectations is an excellent way to produce code that will seem slow either upon deployment or very soon thereafter.

Expectations about speed concern more than just how long it takes to do something. The variation (*jitter*) in the speed is also important. The less variation, the better. One reason for this is pragmatic. If I request an action that takes longer than I'd like, but I know that it almost always takes about the same amount of time (i.e., the speed of the action has low variability), I can plan for the delay and act

accordingly. If I'm an end user and I know that after clicking on "burn CD," it's about 5 minutes before the CD will be done, I can plan to take a break, get a cup of coffee, return a quick phone call, whatever. If, however, the wait is anywhere from 1-10 minutes, it's a lot harder for me to schedule my wait time. The situation is similar if I'm a programmer calling into an API. If I call a function that's comparatively slow, but still of predictable duration (i.e., has low variability), I can plan for the slowness and try to mask it. I can put the call in a part of the program that's not speed-critical, for example, or I can make the call asynchronously, enabling me to proceed with other work while waiting for the requested action to be completed. If the variability is large, however, it's harder for me to cope. If the function often returns quickly (but sometimes returns very slowly), an asynchronous call may not be a viable option, because the overhead of creating a new thread or process may be too large for those common cases where the function returns quickly.

Another reason why low variability is desirable is that it's consistent with the subjective nature of being fast. If an action sometimes runs very quickly but sometimes does not, it's unlikely to be considered "fast." It might be "usually fast" or "often fast," but "fast" alone is usually applied only to things that are reliably and predictably fast. Something (whether an application or an API) with a lot of speed variability is rarely perceived as being fast, regardless of its "usual" or "normal" performance. If your goal is to be fast, you can't avoid the need for low variability.

This has some interesting low-level consequences. Cache misses and page faults, for example (topics I'll return to many times in this book) are anathema to fastware not just because they incur speed penalties in and of themselves, but also because they increase the variability in the speed of memory accesses.

We have seen that whether an action is deemed "fast" is influenced by the perception and expectations of the client requesting the action, but it's fundamentally determined by how long it takes for an action to be performed. The time between requesting an action and that action being performed is known as *latency*, but it's important to distinguish two aspects of latency:

- **Initial latency** corresponds to when the results of an action *begin* to be available. We'll call this L_{init}.
- **Total latency** corresponds to when the results are *completely* available. We'll call this L_{total}.

Consider again the example of a function that finds all documents on a network containing a specified search term and returns the results in a buffer. We sketched two implementations. One – what we'll call the "simple" implementation – found all the documents before returning. The other returned a buffer, initially empty, that was being filled by threads or processes running in the background. We'll call this the "complex" implementation.

For the simple implementation, L_{init} and L_{total} are the same; the entire set of matched documents becomes available at once. For the complex implementation,

however, L_{init} will typically be less than L_{total}. That could be important to clients who want to process the documents serially, because they could get started on the processing sooner. On the other hand, the overhead needed to create the concurrent threads or processes to do the searching and the synchronization needed to allow these multiple agents to simultaneously access the buffer might mean that L_{total} for the complex implementation was greater than L_{total} for the simple implementation. Even if not, the buffer-accessing code for the complex implementation would be more complicated than for the simple implementation (e.g., a request to get the next element from an empty buffer would have to wait for all outstanding searching threads or processes to finish before returning an indication that there were no more elements in the buffer). That, in turn, could mean that the serial processing code's L_{total} would increase, because it would have to access the buffer through the more complex (and presumably slower) interface.

Of course, if the client requesting the documents needed to sort them by length before doing any further processing, only L_{total} would be important (to that client).

In general, fastware is designed to have latency that's as low as possible, but whether the goal is to minimize L_{init} or L_{total} depends on the anticipated usage of the functionality being implemented. In many cases, it's worth offering separate interfaces for the same information, one designed to minimize L_{init}, the other to minimize L_{total}. Databases, for example, tend to offer cursor-based interfaces for iteration that reduce L_{init} at the expense of L_{total}; and also batch-based interfaces that reduce L_{total} at the expense of L_{init}. In this book, "fast" means "low latency." Whether that's L_{init}, L_{total} or both should be apparent from the context of the discussion.

It should come as no surprise that the "low" in "low latency" generally means "as low as possible," because much fastware can never be fast enough.

Latency versus Related Characteristics

When discussing fast software, there are a passel of speed-related terms (other than latency) that commonly arise. Here are how the most important terms relate to fastware.

- **Performance** is a term too general to be terribly useful because, in practice, it tends to mean "does really well at whatever is important to me." It can be a synonym for "fast," but it can just as easily be another word for efficiency, bandwidth, or throughput, each of which is examined below.
- **Efficiency** is concerned with how well you make use of the resources available to you. Software that's highly efficient squeezes every bit of use out of whatever resource is of interest. CPU-efficient software does a lot of work per CPU cycle. Memory-efficient software accomplishes much per byte of memory used. Power-efficient software gets a lot done per watt consumed. Efficient software

might be fast, but efficiency isn't fundamentally concerned with speed, it's concerned with making good use of resources. Pure fastware is willing to risk low resource utilization in exchange for speed, though in practice, fastware developers tend to look at underutilized resources and ponder how they could be exploited to make things faster.

- **Bandwidth** is a measure of how much data can be moved per unit time. As such, it can be a significant factor affecting $L_{total,}$ but it has no effect on L_{init}. The time it takes to completely move data between points (L_{total}) is determined by the initial latency (L_{init}) plus the time to transmit the data (the size of the data divided by the bandwidth). For fastware purposes, bandwidth is typically unimportant compared to latency when small amounts of data are being moved. Many packets on the Internet are small, for example, so the latency in the connections between machines is often more important than the bandwidth. Similarly, when reading small amounts of data from memory, the memory access time (latency) is more important than the bus bandwidth. When large amounts of data are being moved and L_{total} is more important than L_{init} (e.g., large graphics being downloaded for display on a web page), bandwidth becomes the dominant fastware determinant.
- **Throughput** is the computational analogue of bandwidth: it describes how much work can get done per unit time. Like bandwidth, it has no effect on L_{init}, but it can reduce L_{total}, so a high-throughput system may or may not be fast. A superscalar pipelined processor, for example, generally offers higher throughput than its unpipelined cousins, but at the expense of increased initial latency.
- **Scalability** generally refers to the ability of a system to maintain its performance (often latency, responsiveness, or throughput) under increasing load (e.g., larger data sets, more connections, more transactions) by adding additional resources (i.e., more cores or processors, more memory). Systems that scale well may or may not be fast, but they can maintain their speed as workloads increase by having new resources added. Systems that don't scale well often get slower as workloads increase. As a general rule, systems that scale well include some kind of administrative component that monitors performance and makes it possible to adapt to the addition of new resources (e.g., a load balancer for a web farm, code in an extensible hash table to force a rehash when the load factor gets too high). The existence and execution of these administrative components exacts an overhead that would otherwise not be present, so a scalable system may be slower than a unscalable system under light or moderate loads.
- **Responsiveness** is a measure of how quickly a user interface (typically, but not necessarily, a graphical user interface) responds to user actions such as moving a mouse, pressing a button, asking for a file to be opened or saved, etc. It refers to latency (usually L_{total}) in the user interface. Commonly-promul-

gated responsiveness guidelines divide UI actions into three categories, with upper latency limits of $1/10^{th}$ second, 1 second, or 10 seconds. The NetBeans guidelines,† for example, specify that painting of window bars must complete in no more than $1/10^{th}$ second, dialogs must open within 1 second, actions taking longer than 1 second must show some kind of "working" indicator (e.g., a wait cursor), and actions taking longer than 10 seconds must show a progress bar. As we've already seen, delays of longer than $1/10^{th}$ second are contrary to the goal of users having a truly interactive experience.

- **Real-time** software has timing deadlines it must meet, but these deadlines need not be demanding, and once they've been met, there is typically no direct benefit to the real-time system in making the software faster. Software displaying an automobile's speed in its dashboard, for example, need not update faster than about every tenth of a second, a sampling rate that's not hard to program for and that gains nothing by being faster. There is thus no inherent link between "real-time" and "fast." Many developers of real-time embedded systems are effectively producing fastware, but that's typically a side-effect of other goals, e.g., reducing the cost or power consumption of the hardware while still allowing the software to meet its real-time constraints.

Each of the above issues corresponds to a characteristic we'd like to see in our software. Even developers of non-real-time systems have timing deadlines they must meet, because, as we've already noted, all software has a point where it's too slow to be useful. This book focuses on latency, because low latency is the hallmark of fastware, but that doesn't mean efficiency, throughput, scalability, etc. aren't important. They are. Lots of other software characteristics are important, too, including readability, maintainability, extensibility, portability, and "quick developability" -- the ability to develop the software in a reasonable amount of time. (I won't mention correctness, because, again, *speed is a correctness criterion*.)

It is generally impossible to maximize all of these characteristics; trade-offs have to be made. But the best trade-offs involve a full understanding of the options available, and when a trade-off against speed is one of the things being considered, it's advantageous to know as many ways to achieve speed as possible. The purpose of this book is to give you that information. *Fastware!* adopts a single-minded focus on making software fast, and it largely ignores the effect this may have on other characteristics of the system. Not because speed is most important, but because you can't intelligently sacrifice speed for some other characteristic if you're not aware of the many ways speed can be achieved.

---

† *What is UI Responsiveness*, http://performance.netbeans.org/responsiveness/whatisresponsiveness.html, fetched on 8 April 2008.

Producing Fastware

We can think of well-written "normal" software as having a "natural" speed at which it works. Fastware runs faster for two reasons. First, it does fewer things that tend to slow things does, i.e., it avoids common speed impediments. Second, it does more things that tend to speed things up, i.e., it employs more latency-reducing tools and techniques. In this sense, fastware is like race cars compared to normal cars. Race cars differ from normal cars in that they eliminate things that would slow them down (e.g., unnecessary weight, avoidable aerodynamic drag) and they add things that speed them up (e.g., more powerful engines, tires with greater traction). Producing fastware requires a knowledge of what tends to slow things down and what tends to speed things up.

At its essence, the secret to writing fastware was perhaps best summarized by Michael Rauchman:

"Understand your problem and understand your tools."

In many cases, the most important part of this advice is understanding your problem. The most sensational speed improvements often result from deep insights into what you need to do and do *not* need to do, then tailoring your solution to do only what is truly needed. Chapter 4 is devoted to this critical aspect of producing fastware. Everything else in the book is important, but the gains to be made from really understanding what your software needs to do often trump anything you can squeeze out of general-purpose software tricks and hardware exploits.

But I'm getting ahead of myself. Before discussing how to take advantage of a problem's characteristics to make it really scream, we need to cover two topics that crop up repeatedly in fastware discussions. The first is memory, processors, and their interactions. Software runs on hardware, and if you don't understand what's happening with memory and processors, you're working with a big handicap. The information in Chapter 2 will see that you don't.

The second topic is the indispensable role of accurate, empirical speed-related data. Both software and hardware systems have become so complex, predicting what will speed them up, slow them down, or have no effect is often impossible. What succeeds brilliantly in one situation fails disastrously in others; my system's optimization is yours' pessimization. Chapter 3 delves into the whys and hows of gathering empirical data about the speed-related behavior of your systems.

We then turn to problem-specific optimization (Chapter 4); avoiding computation (Chapter 5); choosing among data structures and algorithms (Chapter 6); improving IO speed (Chapter 7); taking advantage of concurrency (Chapter 8); working effectively with runtime memory managers (Chapter 9); and addressing the speed-related behavior of the facets of your systems over which you have, alas, only indirect control (Chapter 10).

The final chapters of the book combine the ideas of everything that's come before them. Chapter 11 focuses on the design and implementation of new systems where speed is an important concern, while Chapter 12 explores — finally! — what usually comes to mind when people think about performance optimization: how to take systems that aren't fast enough and make them faster.

One topic that's not in the book is how to apply your programming language (or languages) so that the software you write runs with maximal speed. That may surprise you. If the key to fastware is understanding your problem and understanding your tools, certainly your most obvious and frequently-used tool is the programming language(s) you use.

Programming languages are powerful and complex, each with its own set of speed-related idioms, and even seemly similar languages such as C and C++ follow different (sometimes contradictory) paths to speed. A rich literature explaining what's fast and what's not has cropped up around every commonly used programming language, and it makes no sense for me to try to summarize that body of information. Instead, I focus on ideas, tools, and techniques that transcend the particulars of individual programming languages, in some cases showing how apparently unrelated language-specific advice stems from more general underlying issues. In C and C++, for example, having functions return large objects is often unacceptably slow, while in C# and Java, the creation of too many transient objects can slow down the garbage collector. The crux of the issue in both cases is the cost of copying things: C and C++ entities are copied when they are returned from functions, and the compacting garbage collectors typically used by C# and Java copy objects during compaction. In *Fastware!*, I focus on things like the cost of unnecessary copying, because that's an important matter, regardless of the programming language you use. I mention common manifestations of issues such as this as they arise in various programming languages, but for specifics of what's fast in various languages and what's not, you'll need to consult language-specific sources.

The fundamental ideas behind fastware are these:

- **Speed is a correctness criterion**. Like all correctness criteria, it's designed in from the start, not stapled on at the end.
- **Know what you really have to do**. Avoid doing what you don't have to do, postpone doing what you only might have to do, and do as much of what you know you have to do in advance.
- **Get others to help you**, e.g., other threads, other machines, other software, other parts of the system you're a part of.
- **Empirically verify all speed improvements**: measure, measure, measure. Some things that "should" improve speed don't, some that "should" hurt speed help it, and what works in one situation may hurt in others.

The Voice of Experience

Know Your Programming Language

Sergey Solyanik
Developer since 1993

I was going through an interview package where the interviewee returned a whole STL vector as a result of the function. I wondered if this is a reasonable thing to do, and whether it would cause a lot of data copying. So I cranked up Visual Studio 2008, jotted down a few lines of code, and looked at the disassembly.

What drew my attention was nearly 30 instructions that it generated for "vector.push_back(i)". I expected that the optimized code would be a lot less, and started digging. As it turned out, it was a lot worse. The code generated inline was the tip of an iceberg - another function was called, and not just in the case where array reallocation was required. I stepped through it in assembly several times to make sure.

I decided to measure how it would compare to non-STL programming idioms. I compared pushing 10000 numbers into pre-allocated STL vector, an auto-expanding STL vector, an expanding C array, and a pre-allocated C array. I restricted the code to a single core, and for a good measure, pre-committed the heap to ensure that paging does not spoil the results. For stats, I calculated the average, the median, and the histogram on the log scale.

The results are quite damning: the most commonly accepted C idiom of pre-allocating the array is 2 clocks per integer inserted, compared to 8 for pre-allocated vector, and 150 for most commonly used dynamically expanding vector. Which is 4 to 75 times(!) slower than C. Even the dynamically expanding C array is the same speed as the pre-allocated vector, and 2.5 times faster than its dynamically-expanding STL counterpart. Which brings up the question - why use C++ at all, if the only advantage it has over Java and C# - the performance - is eaten by the runtime?

Soon after blogging about my experience, reader comments made me realize that my exercise in STL vector perf analysis was to a large extent an exercise in n00bness. Revising my benchmark code showed that C cost from 2 to 70 clocks per iteration, but C++'s range was from 2 to 140 clocks per iteration. The best C++ performance (as good as C) required use of both language-specific idioms (e.g., using reserve to preallocate memory) and implementation-specific idioms (e.g., disabling optional security checks), neither of which I'd originally done.

Conclusion: know what you're doing. Performance can be from 3x to 70x slower if you're not careful.

Sergey Solyanik is a development manager at Microsoft. This Voice of Experience was adapted from his February 2008 blog entries, "STL vector performance" (http://1-800-magic.blogspot.com/2008/02/stl-vector-performance.html) and "STL vector perf redux" (http://1-800-magic.blogspot.com/2008/02/stl-vector-perf-redux.html).

Reporting Bugs and Making Suggestions

I've done my best to fill this book with accurate, practical information, but I know I could have done a better job. What I know less well is how. Is something incor-

rect? Incomplete? Unclear? Misleading? Did I omit a topic I should have included? Did I include something I shouldn't have? If you have a comment about how I could improve *Fastware!*, please let me know: fastware@aristeia.com.

You can always find the current list of suggestions I've received, as well as how the book has changed in response to them, at http://aristeia.com/BookErrata/fastware1e-errata.html.