We appear to be at a major turning point in the way we develop software.
The major processor architectures, from Intel and AMD to Sparc and PowerPC, have run out of room with most of their traditional approaches to boosting CPU performance. Instead of driving clock speeds and straight-line instruction throughput ever higher, they are turning *en masse* to hyperthreading and multicore architectures. That puts us at a fundamental turning point in software development because for years, we've enjoyed a free lunch as faster computers directly made our applications faster too, and that will largely not be true any more. Most of the coming gains won't be picked up directly by the majority of today's applications.
Arguably, the free lunch has already been over for a year or two, only we're just now noticing.

The Free Performance Lunch

There's an interesting phenomenon that's known as "Andy giveth, and Bill taketh away." Make a CPU 10 times as fast, and software usually finds 10 times as much to do (or, in some cases, will feel at liberty to do it 10 times less efficiently). Most classes of applications have enjoyed free and regular performance gains for several decades, even without releasing new versions or doing anything special, because the CPU manufacturers (primarily) and memory and disk manufacturers (secondarily) have reliably enabled ever-newer and ever-faster mainstream systems.
Over the past 30 years, gains in CPU performance have come in three main areas, two of which focus on straight-line execution flow:

- Clock speed (more cycles). Running the CPU faster means doing the same work faster.
- Execution optimization (doing more work per cycle). Optimizations such as pipelining, branch prediction, executing multiple instructions in the same clock cycle(s), and even reordering the instruction stream for out-of-order execution, all make the instructions flow better and/or execute faster.
- Cache. Main memory continues to be so much slower than the CPU that it makes sense to put the data closer to the processor—and you can't get much closer than being right on the die.

A fundamentally important thing to recognize about this list is that all of these areas are concurrency-agnostic. Speedups in any of these areas will directly lead to speedups in existing sequential (nonparallel, single-threaded, single-process) applications, as well as existing applications that do use concurrency. That's important because the vast majority of today's applications are single threaded. Because growth has come in these areas, pretty much all old applications have always run significantly faster on new CPUs, even without being recompiled to take advantage of all the new instructions and features offered by the latest CPUs.

Obstacles, and Why You Don't Have 10 GHz Today

CPU performance growth as we have known it hit a wall two years ago. Most people have only recently started to notice.
You can get similar graphs for other chips, but I'm going to use Intel data here. Figure 1 graphs the history of Intel chip introductions by clock speed and number of transistors. Around the beginning of 2003, you'll note a sharp flattening from the previous trend toward ever-faster CPU clock speeds. It has become harder and harder for CPU designers to exploit higher clock speeds due to several physical issues, notably heat, power consumption, and current leakage problems.
Quick: What's the clock speed on the CPU(s) in your current workstation? Are you running at 10 GHz? We should be, according to CPU trends before 2003, but a quick look around shows that we aren't. We're at 3.4 GHz (on Intel), and even 4 GHz seems to be remote indeed, given that in 2004, Intel first delayed and then abandoned its plans to introduce a 4-GHz chip in the foreseeable future. We'll probably see 4-GHz CPUs in our mainstream desktop machines someday, but we don't know when.

TANSTAAFL: Moore's Law and the Next Generation(s)

> There ain't no such thing as a free lunch.
>
> —R.A. Heinlein
>
> *The Moon Is a Harsh Mistress*

Moore's Law predicts exponential growth, and clearly exponential growth can't continue forever before we reach hard physical limits; light isn't getting any faster. (Caveat: Moore's Law applies principally to transistor densities, but the same kind of exponential growth has occurred in related areas such as MIPS and clock speeds. There's even faster growth in other spaces, most notably the data-storage explosion, but that important trend belongs in a different article.)

So are we at the end of Moore's Law? Not yet. Chip engineers are hitting higher and higher walls and are being forced to detour and exploit gains in different directions, but transistor counts continue to explode (for now) and raw CPU throughput will continue to follow Moore's Law-like growth for some years to come.

The key difference is how all those transistors will be used: The performance gains are going to be accomplished in fundamentally different ways, and most current applications will no longer benefit from the free ride without significant redesign.

For the near-term future, the performance gains in new chips will be fueled by three main approaches, only one of which is the same as in the past:

- Hyperthreading (running two or more threads in parallel inside a single CPU). Hyperthreaded CPUs are now available and do sport extra registers and allow some instructions to run in parallel. It's not the same as having two real CPUs, but hyperthreading is sometimes cited as offering a 5-15 percent performance boost for reasonably well-written multithreaded applications, or even as much as 40 percent under ideal conditions for carefully written multithreaded applications. That's good, but it's hardly double, and it doesn't help single-threaded applications.
- Multicore (running two or more actual CPUs on one chip). The performance gains here are about the same as having a true dual-CPU system, which is better than hyperthreading but is still less than double the speed even in the ideal case, and it will boost reasonably well-written multithreaded applications. Not single-threaded ones.
- Cache. On-die cache sizes can be expected to continue to grow. Of these three areas, only this will broadly benefit existing applications. Because space is speed, this will save some existing applications for a few more years, even as they process more data and add more code for new features, as long as the working set of performance-sensitive operations continues to fit into the bigger caches.

But cache is it: Hyperthreading and multicore CPUs will have nearly no impact on most current applications.

What This Means for Software: The Next Revolution

In the 1990s, we learned to grok objects. That revolution in mainstream software development was the greatest such change in the past 20 (and arguably 30) years. There have been other interesting changes, but nothing that most of us have seen during our careers has been as fundamental and as far reaching a change in the way we write software as the object revolution.

Until now. Starting today, the performance lunch isn't free any more. Sure, there will continue to be generally applicable performance gains that everyone can pick up, thanks mainly to cache sizes. But if you want your application to benefit from the continued exponential throughput advances in new processors, it will need to be a well-written, concurrent (usually multithreaded) application.

I can hear the howls of protest: "Concurrency? That's not news! People are already writing concurrent applications." That's true. Of a small fraction of developers.

Remember that people have been doing object-oriented programming since at least the days of Simula in the late 1960s. But OO didn't become a revolution, and dominant in the mainstream, until the 1990s, when our industry was driven by requirements to write larger and larger systems that solved larger and larger problems and exploited the greater and greater CPU and storage resources that were becoming available. OOP's strengths in abstraction and dependency management made it a necessity for achieving large-scale software development that is economical, reliable, and repeatable. OO was ready, and when the time was right, it exploded into dominance.

Similarly, we've been doing concurrent programming since those same dark ages, writing coroutines and monitors and similar jazzy stuff. And for the past decade or so, we've witnessed incrementally more and more programmers writing concurrent systems. But an actual revolution marked by a major turning point toward concurrency has been slow to materialize. Today, the vast majority of applications are single threaded.

Concurrency is now ready, and it will explode into dominance; it is the next major revolution in how we write software. Different experts still have different opinions on whether it will be bigger than OO, but that kind of conversation is best left to pundits. For technologists, the interesting thing is that concurrency is of the same order as OO both in the (expected) scale of the revolution and in the complexity and learning curve of the technology.

Benefits and Costs of Concurrency

Today, when we're using concurrency already, it's generally for two major reasons: a) for clarity, to logically separate naturally independent control flows; and/or b) for performance, either to scalably take advantage of multiple physical CPUs or to easily take advantage of latency in other parts of the application.

There are, however, real costs to concurrency. Some of the obvious costs are actually relatively unimportant. For example, yes, locks can be expensive to acquire, but when used judiciously and properly, you gain much more from the concurrent execution than you lose on the synchronization, if you can find a sensible way to parallelize the operation and minimize or eliminate shared state.

Perhaps the second-greatest cost of concurrency is that not all applications are amenable to parallelization.

Probably the greatest cost of concurrency is that it really is hard: The programming model, meaning the model in the programmer's head that he needs to reason reliably about his program, is much harder than it is for sequential control flow. Just as it is a leap for a structured programmer to learn OO (What's an object? What's a virtual function? How should I use inheritance? And beyond the "whats" and "hows," why are the correct design practices actually correct?), it's a leap of about the same magnitude for a sequential programmer to learn concurrency (What's a race? What's a deadlock? How can it come up, and how do I avoid it? What constructs actually serialize the program that I thought was parallel? And beyond the "whats" and "hows," why are the correct design practices actually correct?).

The vast majority of programmers today don't grok concurrency, just as the vast majority of programmers 15 years ago didn't yet grok objects. But the concurrent programming model is learnable, particularly if we stick to lock-based programming, and once grokked, it isn't that much harder than OO and hopefully can become just as natural. Just be ready and allow for the investment in training and time, for you and for your team.

(I deliberately limited this discussion to lock-based concurrent programming models. There is also lock-free programming, supported most directly at the language level in Java 5 and in at least one popular C++ compiler. But concurrent lock-free programming is extremely hard for programmers to understand and reason about. Most of the time, only systems and library writers should have to understand lock-free programming, although virtually everybody should be able to take advantage of the lock-free systems and libraries those people produce.)

What It Means for Us

Okay, back to what it means for us.

1. The clear primary consequence (which we've already covered) is that applications will increasingly need to be concurrent if they want to fully exploit future CPU throughput gains. "Oh, performance doesn't matter so much, computers just keep getting faster" has always been a naive statement to be viewed with suspicion, and for the near future, it will be simply wrong for most of today's applications. A significant challenge here is that not all software problems are amenable to parallelization.
2. Applications are likely to become increasingly CPU-bound. We seem to have reached the end of the "applications are increasingly I/O-bound or network-bound or database-bound" trend, because performance in those areas is still improving rapidly (gigabit WiFi, anyone?) while traditional CPU performance-enhancing techniques have maxed out. As we continue to demand that programs handle vastly more data and add more features, we will increasingly find that the programs run out of CPU to do it all, unless we code for concurrency.

There are two ways to deal with this sea change toward concurrency. One is to redesign your applications for concurrency, as above. The other is to be frugal, by writing code that is more efficient and less wasteful. This leads to the third interesting consequence:

1. Efficiency and performance optimization will get more, not less, important. Those languages, such C and C++, that already lend themselves to heavy optimization will find new life (and that's on top of the fact that C is alive and well, and that C++ has already continued to grow steadily anyway). Those that don't will need to find ways to compete and become more efficient and optimizable. Expect long-term increased demand for performance-oriented languages and systems.
2. Finally, programming languages and systems will increasingly be forced to deal well with concurrency. Java has included support for concurrency since its beginning, although mistakes were made that later had to be corrected over several releases in order to do concurrent programming more correctly and efficiently. C++ has long been used to write heavy-duty multithreaded systems well, but it has no standardized support for concurrency at all (the ISO C++ Standard doesn't even mention threads, and does so intentionally), and so typically, the concurrency is of necessity, accomplished by using nonportable, platform-specific concurrency features and libraries. (It's also often incomplete; for example, static variables must be initialized only once, which typically requires that the compiler wrap them with a lock, but many C++ implementations do not generate the lock.)

Conclusion

If you haven't done so already, now is the time to take a hard look at the design of your application, determine what operations are CPU-sensitive now or are likely to become so soon, and identify how those places could benefit from concurrency. Now is also the time for you and your team to grok concurrent programming's requirements, pitfalls, styles, and idioms.

A few rare classes of applications are naturally parallelizable, but most aren't. Even when you know exactly where you're CPU-bound, you may well find it difficult to figure out how to parallelize those operations; all the more reason to start thinking about it now. Implicitly parallelizing compilers can help a little, but don't expect much; they can't do nearly as good a job of parallelizing your sequential program as you could do by turning it into an explicitly parallel and threaded version.

Thanks to continued cache growth and probably a few more incremental straight-line control flow optimizations, the free lunch will continue a little while longer; but starting today, the buffet will only be serving that one entrée and that one dessert. The filet mignon of throughput gains is still on the menu, but now it costs extra-extra development effort, extra code complexity, and extra testing effort. The good news is that for many classes of applications, the extra effort will be

worthwhile because concurrency will let them fully exploit the continuing exponential gains in processor throughput.