DOI:10.1145/1506409.1506425

Multicore computers shift the burden of software performance from chip designers and processor architects to software developers.

BY JAMES LARUS

Spending Moore's Dividend

OVER THE PAST three decades, regular, predictable improvements in computers have been the norm, progress attributable to Moore's Law, the steady 40%-per-year increase in the number of transistors per chip unit area.

The Intel 8086, introduced in 1978, contained 29,000 transistors and ran at 5MHz. The Intel Core 2 Duo, introduced in 2006, contained 291 million transistors and ran at 2.93GHz.[9] During those 28 years, the number of transistors increased by 10,034 times and clock speed 586 times. This hardware evolution made all kinds of software run much faster. The Intel Pentium processor, introduced in 1995, achieved a SPECint95 benchmark score of 2.9, while the Intel Core 2 Duo achieved a SPECint2000 benchmark score of 3108.0, a 375-times increase in performance in 11 years.[a]

a Benchmarks from the 8080 era look trivial today and say little about modern processor performance. A realistic comparison over the decades requires a better starting point than the 8080. Moreover, the revision of the SPEC benchmarks every few years frustrates direct comparison. This comparison normalizes using the Dell Precision WorkStation 420 (800MHz PIII) that produced 364 SPECint2000 and 38.9 SPECint95, a ratio of 9.4.

These decades are also when the personal computer and packaged software industries were born and matured. Software development was facilitated by the comforting knowledge that every processor generation would run much faster than its predecessor. This assurance led to the cycle of innovation outlined in Figure 1. Faster processors enabled software vendors to add new features and functionality to software that in turn demanded larger development teams. The challenges of constructing increasingly complex software increased demand for higher-level programming languages and libraries. Their higher level of abstraction contributed to slower code and, in conjunction with larger and more complex programs, drove demand for faster processors and closed the cycle.

This era of steady growth of single-processor performance is over, however, and the industry has embarked on a historic transition from sequential to parallel computation. The introduction of mainstream parallel (multicore) processors in 2004 marked the end of a remarkable 30-year period during which sequential computer performance increased 40%–50% per year.[4] It ended when practical limits on power dissipation stopped the continual increases in clock speed, and a lack of exploitable instruction-level parallelism diminished the value of complex processor architectures.

Fortunately, Moore's Law has not been repealed. Semiconductor technology still doubles the number of transistors on a chip every two years.[7] However, this flood of transistors is now used to increase the number of independent processors on a chip, rather than to make an individual processor run faster.

The challenge the computing industry faces today is how to make parallel computing the mainstream method for improving software performance. Here, I look at this problem by asking how software consumed previous

Advanced Micro Devices multiple 45nm quad core die based on the Opteron processor, codenamed "Shanghai" (www.amd.com/)

PHOTOGRAPH COURTESY OF AMD

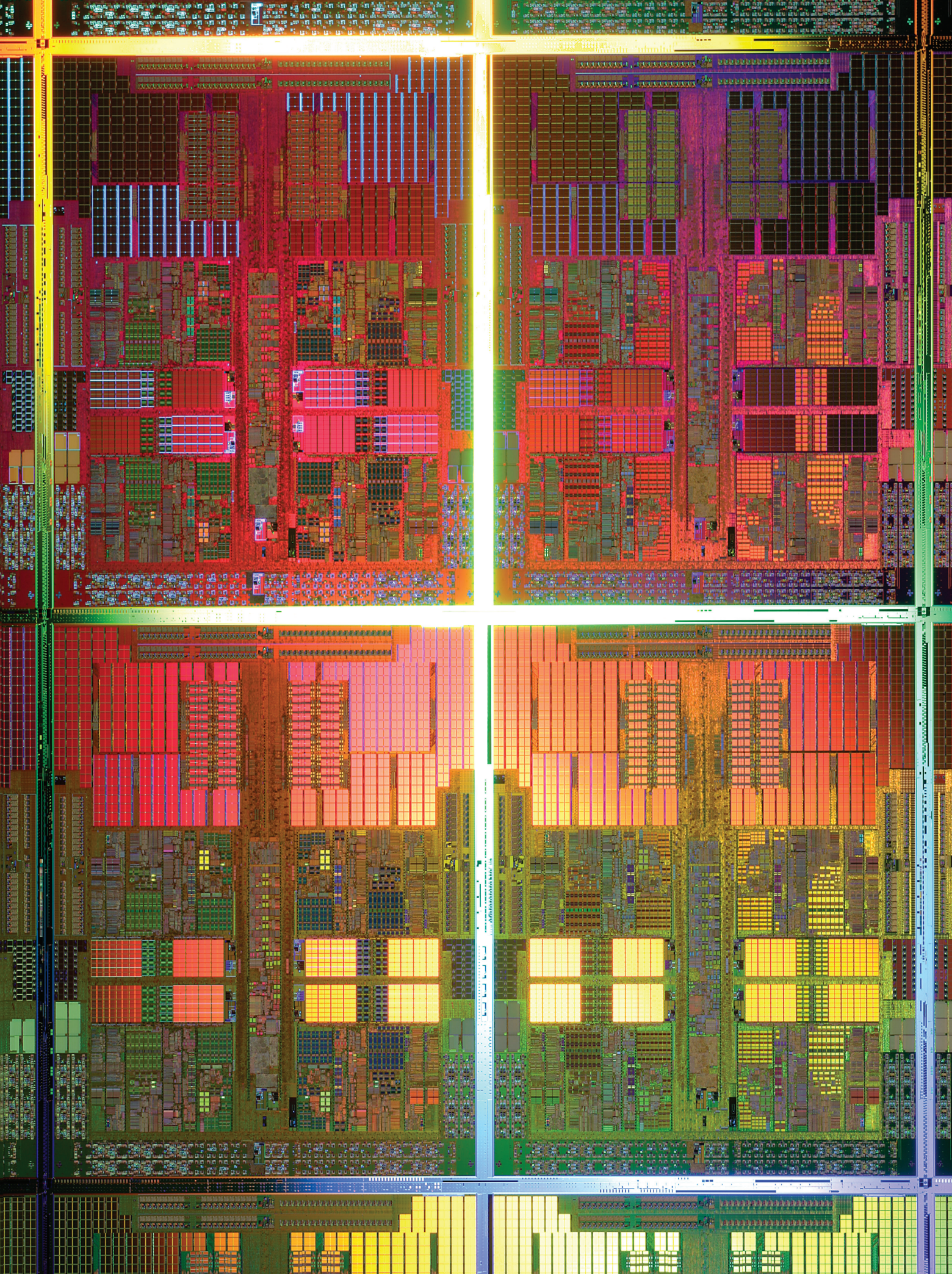


Figure 1: Cycle of innovation in the computer industry.

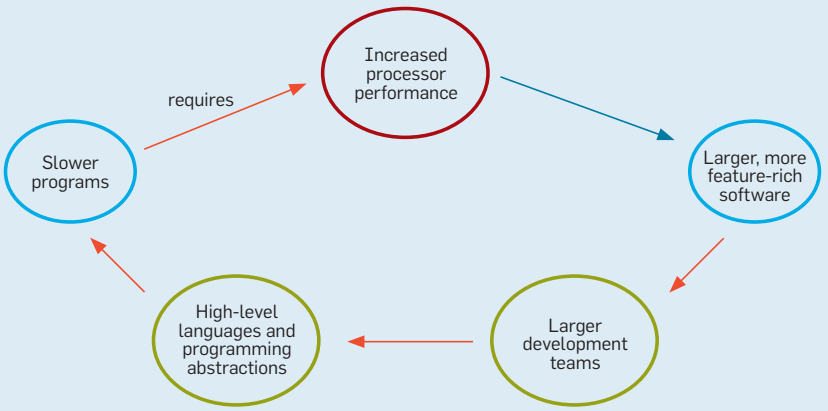
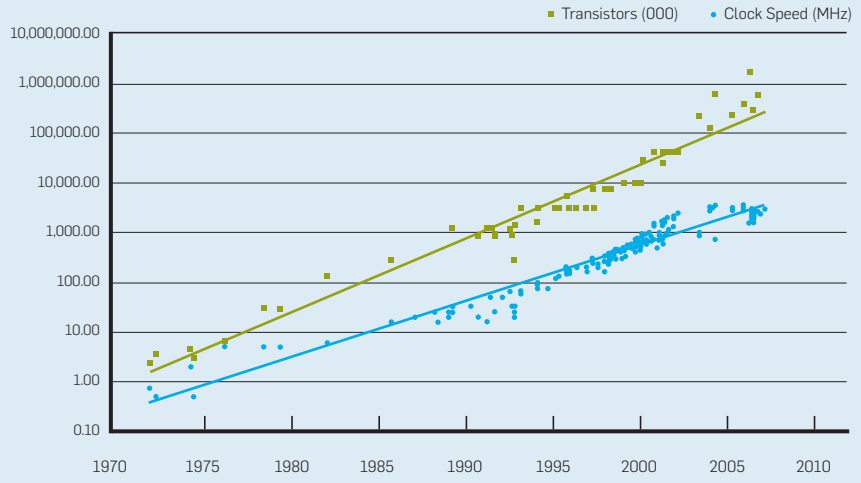


Figure 2: Improvement in Intel x86 processors; data from Olukotun,[18] Herb Sutter, a principal architect at Microsoft, and Intel.



processor-performance growth and whether multicore processors can satisfy the same needs. In short, how did we use the benefits of Moore's Law? Will parallelism continue the cycle of software innovation?

In 1965, Gordon Moore, a co-founder of Intel, postulated that the number of transistors that could be fabricated on a semiconductor chip would double every year,[17] a forecast he subsequently reduced to every second year.[10] Amazingly, this prediction still holds. Each generation of transistor is smaller and switches at a faster speed, allowing clock speed (and computer performance) to increase at a similar rate. Moreover, abundant transistors enabled architects to improve processor design by implementing sophisticated microarchitectures. For convenience, I call this combination of improvements in computers Moore's Dividend. Figure 2 depicts the evolution of Intel's x86 processors. The number of transistors in a processor increased at the rate predicted by Moore's Law, doubling every 24 months while clock frequency grew at a slightly slower rate.

These hardware improvements increased software performance. Figure 3 charts the highest SPEC integer benchmark score reported each month for single-processor x86 systems. Over a decade, integer processor performance increased by 52 times its former level.

Myhrvold's Laws

A common belief among software developers is that software grows at least at the same rate as the platform on which it runs. Nathan Myhrvold, former chief technology officer at Microsoft, memorably captured this wisdom with his four laws of software, following the premise that "software is a gas" due to its tendency to expand to fill the capacity of any computer (see the sidebar "Nathan Myhrvold's Four Laws of Software").

Support for this belief depends on the metric for the "volume" of software. Soon after Myhrvold published the "laws," the rate of growth of lines of code (LoC) in Windows diverged dramatically from the Moore's Law curve (see Figure 4). This makes sense intuitively; a software system might grow quickly in its early days, as basic functionality accrues, but exponential growth (such as the factor-of-four increase in lines of code between Windows 3.1 and Windows 95 over three years) is difficult to sustain without a similar increase in developer headcount or remarkable—unprecedented—improvement in software productivity.

Software volume is also measured in other ways, including necessary machine resources (such as processor speed, memory size, and capacity). Figure 5 outlines the recommended resources suggested by Microsoft for several versions of Windows. With the exception of disk space (which has increased faster than Moore's Law), the recommended configurations grew at roughly the same rate as Moore's Law.

How could software's resource requirements grow faster than its literal size (in terms of LoC)? Software changed and improved as computers became more capable. To most of the world, the real dividend of Moore's Law, and the reason to buy new computers, was this improvement, which enabled software to do more tasks and do them better than before.

How Was It Spent?

Determining where and how Moore's Dividend was spent is difficult for a number of reasons. Software evolves over a long period, but no one systematically measures changing resource consumption. It is possible to compare released systems, but many aspects of a system or application evolve between releases and without close investigation, and it is difficult to attribute visible differences to a particular factor. Here, I present some experimental hypotheses that await further research

to quantify their contributions to the overall computing experience:

Increased functionality. One of the clearest changes in software over the past 30 years has been a continually increasing baseline of expectations of what a personal computer can and should do. The changes are both qualitative and quantitative, but their cumulative effect has been steady growth in the computation needed to accomplish a specific task.

Software developers will tell you that improvement is continual and pervasive throughout the lifetime of software; new features extend it and, at the same time, raise its computational requirements. Consider the Windows print spooler, with a design that is still similar to Windows 95. Why does it not run 50 times faster today? Oliver Foehr,[5] a developer at Microsoft, analyzed it in 2008 and estimated the consequences of its evolution:

- Additional code over the years added new functionality, most notably improved security and notification, that affected performance by 1.5–4 times, depending on the scenario;
- Printer drivers added functionality for color management and improved treatment of text, graphics, and bookkeeping for a performance effect of a factor of 2;
- Printer resolution and color depth improved from 300*300 dpi at one bit per pixel to at least 600*600 dpi at 24 bits per pixel, or from 1MB to 96MB of image; and
- Memory latency and bandwidth did not keep up with processor speed; the spooler has poor locality due to large color lookup tables and graphics rendering, so its performance was slowed by the increased processor-memory gap.

Software rarely shrinks. Features are rarely removed, since it is difficult to ensure that no customers are still using them. Support for legacy compatibility ensures that the tide of resource requirements always rises and never recedes.

Large-scale, pervasive changes can affect overall system performance. Attacks of various sorts have led programmers to be more careful in writing low-level, pointer-manipulating code, forcing them to take extra care scrutinizing input data. Secure code requires more computation. One in-

Figure 3: Performance improvement in single-processor (x86) SPEC benchmarks (data from www.spec.org); the SPECint95 and SPECint2006 benchmark scores are normalized against SPECint2000.

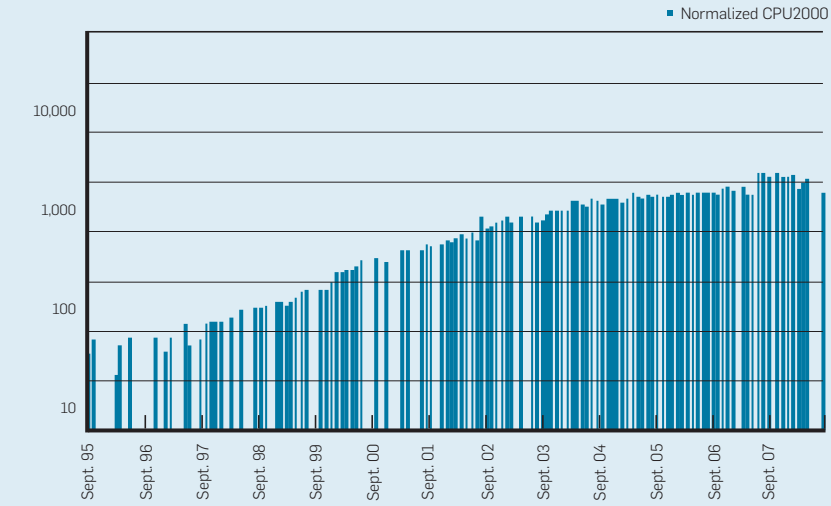


Figure 4: Windows code size (LoC) and Intel processor performance. Code size estimates are from various sources.[13–15]

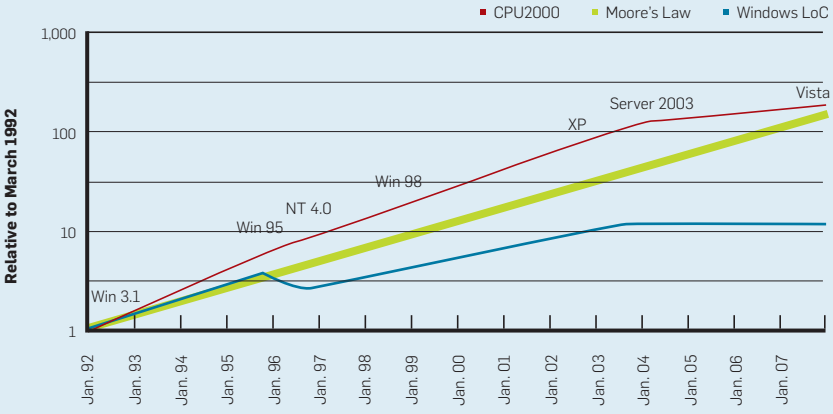


Figure 5: Recommended Windows configurations (maximum values from support.microsoft.com).

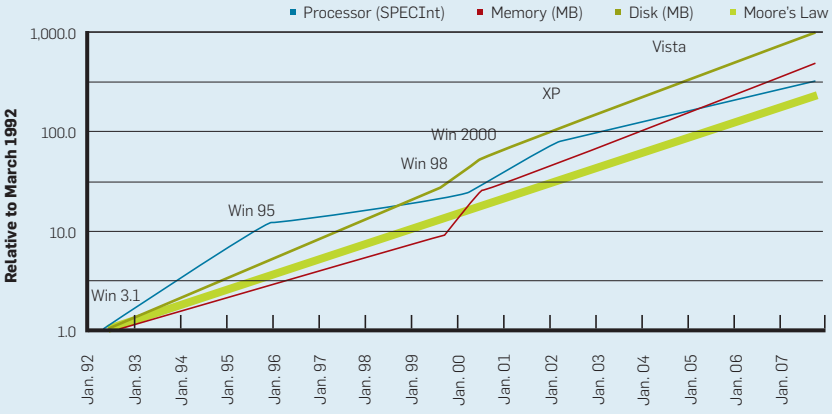


Table 1: Object-oriented complexity metrics (per binary); from an internal Microsoft Research document by Murphy, B. and Nagappan, N. Characterizing Vista Development, December 15, 2006.

	Vista/Win 2003 (Mean per binary)
Total Functions	1.45
Max Class Methods	1.22
Total Class Methods	1.59
Max Inheritance Depth	1.33
Total Inheritance Depth	1.54
Max Subclasses	3.87
Total Subclasses	2.27

dication is that array bounds and null pointer checks impose a time overhead of approximately 4.5% in the Singularity OS.[1] Also important, and equally difficult to measure, are the performance consequences of improved software-engineering practices (such as layering software architecture and modularizing systems to improve development and allow subsets).

Meanwhile, the data manipulated by computers is also evolving, from simple ASCII text to larger, structured objects (such as Word and Excel documents), to compressed documents (such as JPEG images), and more recently to space- and computation-inefficient formats (such as XML). The growing popularity of video introduces yet another format that is even more computationally expensive to manipulate.

Programming changes. Over the past 30 years, programming languages have evolved from assembly language and C code to increased use of higher-level languages. A major step was C++, which introduced object-oriented mechanisms (such as virtual-method dispatch). C++ also introduced abstraction mechanisms (such as classes and templates) that made possible rich libraries (such as the Standard Template Library). These language mechanisms required non-trivial, opaque runtime implementations that could be expensive to execute but improved software development through modularity, information hiding, and increased code reuse. In turn, these practices enabled the construction of ever-larger and more complex software.

Table 1 compares several key object-oriented complexity metrics between Windows 2003 and Vista, showing increased use of object-oriented features. For example, the number of classes per binary component increased 59% and the number of subclasses per binary 127% between the two systems.

These changes could have performance consequences. Comparing the SPEC CPU2000 and CPU2006 benchmarks, Kejariwal et al.[12] attributed the lower performance of the newer suite to increased complexity and size due to the inclusion of six new C++ benchmarks and enhancements to existing programs.

Safe, managed languages (such as C# and Java) further increased the level of programming by introducing garbage collection, richer class libraries (such as .NET and the Java Class Library), just-in-time compilation, and runtime reflection. All these features provide powerful abstractions for developing software but also consume memory and processor resources in nonobvious ways.

Language features can affect performance in two ways: The first is that a mechanism can be costly, even when not being used. Program reflection, a well-known example of a costly language feature, requires a runtime system to maintain a large amount of metadata on every method and class, even if the reflection features are not invoked. The second is that high-level languages hide details of a machine beneath a more abstract programming model. This leaves developers less aware of performance considerations and less able to understand and correct problems.

Mitchell et al.[16] analyzed the conversion of a date object in SOAP format to a Java Date object in IBM's Trade benchmark, a sample business application built on IBM Websphere. The conversion entailed 268 method calls and allocation of 70 objects. Jann et al.[11] analyzed this benchmark on consecutive implementations of IBM's POWER architecture, observing that "modern e-commerce applications are increasingly built out of easy-to-program, generalized but nonoptimized software components, resulting in substantive stress on the memory and storage subsystems of the computer."

I conducted simple programming experiments to compare the cost of implementing the archetypical Hello World program using various languages and features. Table 2 compares C and C# versions of the program, showing the latter has a working set 4.7–5.2 times larger. Another experiment measured the cost of displaying the string "Hello World" by both writing it to a console window and displaying it in a pop-up window. Table 3 shows that a dialog box is 20.7 times computationally more costly in C++ (using Microsoft Foundation Class) and 30.6 times more costly in C# (using Windows Forms). By comparison, the choice of language and runtime system made relatively little difference, as C# was only 1.5 times more costly than C++ for the console and 2.2 times more costly with a window.

This disparity is not a criticism of C#, .NET, or window systems; the

Table 2: Hello World benchmark running on Intel x86, Vista Enterprise, and Visual Studio 2008.

	Debug Build		Optimized Build	
Language	**Working Set**	**Startup Bytes**	**Working Set**	**Startup Bytes**
C	1,424K	6,162	1,304K	5,874
C++	6,756K	113,280	6,748K	87,62

Table 3: Execution cost of displaying "Hello World" string.

Mechanism	Timer Cycles (280ns)
C++, console	1,760
C++, window	36,375
C#, console	2.628
C#, window	80,348

overhead comes with a system that provides a much richer set of functionality that makes programming (and use) of computers faster and less error-prone. Moreover, the cost increases are far less than the performance improvement between the computers of the 1970s and 1980s—when C began—and today.

Decreased programmer focus. Abundant machine resources have allowed programmers to become complacent about performance and less aware of resource consumption in their code. Bill Gates 30 years ago famously changed the prompt in Altair Basic from "READY" to "OK" to save 5B of memory.[6] It is inconceivable today that a developer would be aware of such detail, let alone concerned about it, and rightly so, since a change of this magnitude is unnoticeable.

More significant, however, is a change in the developer mind-set that makes developers less aware of the resource requirements of the code they write:

Increased computer resources means fewer programs push the bounds of a computer's capacity or performance; hence many programs never receive extensive performance tuning. Donald Knuth's widely known dictum "premature optimization is the root of all evil" captures the typical practice of deferring performance optimization until code is nearly complete. When code performs acceptably on a baseline platform, it may still consume twice the resources it might require after further tuning. This practice ensures that many programs run at or near machine capacity and consequently helps guarantee that Moore's Dividend is fully spent at each new release;

Large teams of developers write software. The performance of a single developer's contribution is often difficult to understand or improve in isolation; that is, performance is not a modular property of software. Moreover, as systems become more complex, incorporate more feedback mechanisms, and run on less-predictable hardware, developers find it increasingly difficult to understand the performance consequences of their own decisions. A problem that is everyone's responsibility is no one's responsibility;

The performance of computers is increasingly difficult to understand. It used to suffice to count instructions alone to estimate code performance. As caches became more common, instruction and cache miss counts could identify program hot spots. However, latency-tolerant, out-of-order architectures require a far more detailed understanding of machine architecture to predict program performance; and

Programs written in high-level languages depend on compilers to achieve good performance. Compilers generate good code on average but are oblivious to major performance bottlenecks (such as disks and memory systems) and cannot fix fundamental flaws (such as bad algorithms).

This discussion is not a rejection of today's development practices. There is no way anyone could produce today's software using the artisan, handcraft practices that were possible and necessary for machines with 4K of memory. Moore's Dividend reduced the cost of running a program but increased the cost of developing one by encouraging ever-larger and more complex systems. Modern programming practices, starting with higher-level languages and rich libraries, counter this pressure by sacrificing runtime performance for reduced development effort.

Multicore and the Future

Anyone reading this is able to cite other scenarios in which Moore's Dividend was spent, but in the absence of further investigation and evidence, let's stop and examine the implications of these observations for future software and parallel computers:

Software evolution. Consider the normal process of software evolution, extension, and enhancement in sequential systems and applications. Sequential in this case excludes code running on parallel computers (such as databases, Web servers, scientific applications, and games) that presumably will continue to exploit parallelism on multicore processors.

Suppose a new product release adds functionality that uses a parallel algorithm to solve a computationally demanding task. Developing a parallel algorithm is a considerable challenge, but many problems (such as video processing, natural-language interaction, speech recognition, linear and nonlinear optimization, and machine learn-

Nathan Myhrvold's

Four Laws of Software

Nathan Myhrvold, a former astrophysicist, then Microsoft CTO, explained the dynamics of the computer and software industries as a natural consequence of his observation that software, like a gas, expands to fill its container (research.microsoft.com/acm97/nm/tsld026.htm) in the following ways:

SOFTWARE IS A GAS!
Windows NT lines of code (doubling time 866 days, growth rate 33.9% per year)
Browser Code Growth (doubling time 216 days, growth rate 221% per year)

SOFTWARE GROWS UNTIL IT BECOMES LIMITED BY MOORE'S LAW
Initial growth is quick, like gas expanding (like a browser)
Eventually limited by hardware (like NT)
Brings any processor to its knees, just before the new model is out

SOFTWARE GROWTH MAKES MOORE'S LAW POSSIBLE
That's why people buy new hardware, economic motivator
That's why chips get faster at the same price, not cheaper
Will continue as long as there is opportunity for new software

IMPOSSIBLE TO HAVE ENOUGH
New algorithms
New applications and new users
New notions of what is cool

ing) are computationally intensive. If computational speed inhibits adoption of these techniques—and parallel algorithms exist or can be developed—then multicore processors can enable the addition of compelling new functionality to applications.

Multicore processors are not a magic elixir, just another way to turn additional transistors into more performance. A problem solved with a multicore computer would also be solvable on a conventional processor—if sequential performance had continued its exponential increase. Moreover, multicore does not increase the rate of performance improvement, aside from one-time architectural shifts (such as replacing a single complex processor with a much larger number of simple cores).

New software features that successfully exploit parallelism differ from the evolutionary features added to most software written for conventional uniprocessor-based systems. A feature may benefit from parallelism if its computation is large enough to consume the processor for a significant amount of time, a characteristic that excludes incremental software improvements, small but pervasive software changes, and many simple program improvements.

Using parallel computation to implement a feature may not speed up an application as a whole due to Amdahl's Law's strict connection between the fraction of sequential execution and possible parallel speedup.[8] Eliminating sequential computation in the code for a feature is crucial, because even small amounts of serial execution can render a parallel machine ineffective.

An alternative use for multicore processors is to redesign a sequential application into a loosely coupled or asynchronous system in which computations run on separate processors. This approach uses parallelism to improve software architecture or responsiveness, rather than performance. For example, it is natural to separate monitoring and introspection features from program logic. Running these tasks on a separate processor can reduce perturbation of the mainline computation. Alternatively, extra processors can perform speculative computations to help minimize response time. These uses of parallelism are unlikely to scale with Moore's Law, but giving an application (or portions of an application) exclusive access to a set of processors might produce a more responsive system.

Applications that stop scaling with Moore's Law, either because they lack sufficient parallelism or because their developers no longer rewrite them, will be evolutionary dead ends.

Functionality that does not fit these patterns will not benefit from multicore; rather, such functionality will remain constrained by the static performance of a single processor. In the best case, the performance of a processor may continue to improve at a significantly slower rate (optimistic estimates range from 10% to 15% per year). But in some multicore chips, processors will run slower, as chip vendors simplify individual cores to lower power consumption and integrate more cores.

For many applications, most functionality is likely to remain sequential. For software developers to find the resources to add or change features, it may be necessary to eliminate old features or reduce their resource consumption. A paradoxical consequence of multicore is that sequential performance tuning and code-restructuring tools are likely to be increasingly important. Another likely consequence is that software vendors will be more aggressive in eliminating old or redundant features, making space for new code.

The regular growth in multicore parallelism poses an additional challenge to software evolution. Kathy Yelick, a professor of computer science at the University of California, Berkeley, has said that the experience of the high-performance computing community is that each decimal order of magnitude increase in parallelism requires a major redesign and rewrite of parallel code.[20] Multicore processors are likely to come into widespread use at the cusp of the first such change ($8 \rightarrow 16$); the next one ($64 \rightarrow 128$) is only three processor generations (six years) later. This observation is relevant only to applications that use scalable algorithms requiring large numbers of processors. Applications that stop scaling with Moore's Law, because they lack sufficient parallelism or their developers no longer rewrite them, are performance dead ends.

Parallelism will also force major changes in software development. Moore's Dividend enabled a shift to higher-level languages and libraries. The pressures driving this trend will not change, because increased abstraction helps improve security, reliability,

and program productivity. In the best case, parallelism enables new implementations of languages and features; for example, parallel garbage collectors reduce the pause time of computational threads, thereby enabling the use of safe languages in applications with real-time constraints.

Another approach that trades performance for productivity is to hide the underlying parallel implementation. Domain-specific languages and libraries can provide an implicitly parallel programming model that hides parallel programming from most developers, who instead use abstractions with semantics that do not change when running in parallel. For example, Google's MapReduce library utilizes a simple, well-known programming paradigm to initiate and coordinate independent tasks; equally important, it hides the complexity of running these tasks across a large number of computers.[3] The language and library implementers may struggle with parallelism, but other developers benefit from multicore without having to learn a new programming model.

Parallel software. Another major category of applications and systems already take advantage of parallelism; the two most notable examples are servers and high-performance computing, each providing different but important lessons to systems developers.

Servers have long been the main commercially successful type of parallel system. Their "embarrassingly parallel" workload consists of mostly independent requests that require little or no coordination and share little data. As such, it is relatively easy to build a parallel Web server application, since the programming model treats each request as a sequential computation. Building a Web site that scales well is an art; scale comes from replicating machines, which breaks the sequential abstraction, exposes parallelism, and requires coordinating and communicating across machine boundaries.

High-performance computing followed a different path that used parallel hardware because there was no alternative with comparable performance, not because scientific and technical computations are especially well suited to parallel solution. Parallel hardware is a tool for solving problems. The popular programming models—MPI and OpenMP—are performance-focused, error-prone abstractions that developers find difficult to use. More recently, game programming emerged as another realm of high-performance computing, with the same attributes of talented, highly motivated programmers spending great effort and time to squeeze the last bit of performance from complex hardware.[19]

If parallel programming is to be a mainstream programming model, it must follow the path of servers, not of high-performance computing. One alternative paradigm for parallel computing "Software as a Service" delivers software functionality across the Internet and revisits timesharing by executing some or all of an application on a shared server in the "cloud."[2] This approach to computing, like servers in general, is embarrassingly parallel and benefits directly from Moore's Dividend. Each application instance runs independently on a processor in a server. Moore's Dividend accrues directly to the service provider, even if the application is sequential. Each new generation of multicore processors halves the number of computers needed to serve a fixed workload or provide the headroom needed to add features or handle greater workloads. Despite the challenges of creating a new software paradigm and industry, this model of computation is likely to be popular, particularly for applications that do not benefit from multicore.

Conclusion

Moore's Dividend was spent in many ways and places, ranging from programming languages, models, architectures, and development practices, up through software functionality. Parallelism is not a surrogate for faster processors and cannot directly step into their roles. Multicore processors will change software as profoundly as previous hardware revolutions (such as the shift from vacuum tubes to transistors or transistors to integrated circuits) radically altered the size and cost of computers, the software written for them, and the industry that produced and sold the hardware and software. Parallelism will drive software in new directions (such as computationally intensive, game-like interfaces or services provided by the cloud) rather than continuing the evolutionary improvements made familiar by Moore's Dividend.

Acknowledgments

Many thanks to Al Aho (Columbia University), Doug Burger (Microsoft), David Callahan (Microsoft), Dennis Gannon (Microsoft), Mark Hill (University of Wisconsin), and Scott Wadsworth (Microsoft) for helpful comments and to Oliver Foehr (Microsoft) and Nachi Nagappan (Microsoft) for assistance with Microsoft data. C

References

1. Aiken, M., Fähndrich, M., Hawblitzel, C., Hunt, G., and Larus, J.R. Deconstructing process isolation. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (San Jose, CA, Oct.). ACM Press, New York, 2006, 1–10.
2. Carr, N. *The Big Switch: Rewiring the World, From Edison to Google*. W.W. Norton, New York, 2008.
3. Dean, J. and Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.
4. Ekman, M., Warg, F., and Nilsson, J. An in-depth look at computer performance growth. *ACM SIGARCH Computer Architecture News 33*, 1 (Mar. 2005), 144–147.
5. Foehr, O. personal email communications, June 30, 2008.
6. Gates, B. Personal email (Apr. 10, 2008).
7. Hachman, M. Intel's Gelsinger predicts Intel Inside everything. *PC Magazine* (July 3, 2008).
8. Hill, M.D. and Marty, M.R. Amdahl's Law in the multicore era. *IEEE Computer 41*, 7 (July 2008), 33–38.
9. Intel. The Evolution of a Revolution. Santa Clara, CA, 2008; download.intel.com/pressroom/kits/IntelProcessorHistory.pdf.
10. Intel. *Excerpts from A Conversation with Gordon Moore: Moore's Law.* Video transcript, Santa Clara, CA, 2005; ftp://download.intel.com/museum/Moores_Law/Video-Transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf.
11. Jann, J., Burugula, R.S., Dubey, N., and Pattnaik, P. End-to-end performance of commercial applications in the face of changing hardware. *ACM SIGOPS Operating Systems Review 42*, 1 (Jan. 2008), 13–20.
12. Kejariwal, A., Hoflehner, G.F., Desai, D. Lavery, D.M., Nicolau, A., and Veidenbaum, A.V. Comparative characterization of SPEC CPU2000 and CPU2006 on Itanium architecture. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (San Diego, CA, June). ACM Press, New York, 2007, 361–362.
13. Lohr, S. and Markoff, J. Windows is so slow, but why? *New York Times* (Mar. 27, 2006); www.nytimes.com/2006/03/27/technology/27soft.html.
14. Maraia, V. *The Build Master: Microsoft's Software Configuration Management Best Practices.* Addison-Wesley, Upper Saddle River, NJ, 2006.
15. McGraw, G. *Software Security: Building Security In.* Addison-Wesley Professional, Boston, MA, 2006.
16. Mitchell, N., Sevitsky, G., and Srinivasan, H. The diary of a datum: An approach to analyzing runtime complexity in framework-based applications. In *Proceedings of the Workshop on Library-Centric Software Design* (San Diego, CA, Oct.). ACM Press, New York 2005, 85–90.
17. Moore, G.E. Cramming more components onto integrated circuits. *Electronics 38*, 8 (Apr. 1965), 56–59.
18. Olukotun, K. and Hammond, L. The future of microprocessors. *ACM Queue 3*, 7 (Sept. 2005), 26–29.
19. Sweeney, T. The next mainstream programming language: A game developer's perspective. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, SC, Jan.). ACM Press, New York, 2006, 269–269.
20. Yelick, K. AltTAB. Discussion on parallelism at Microsoft Research, Redmond, WA, July 19, 2006.

James Larus (larus@microsoft.com) is Director of Software Architecture in the Cloud Computing Futures project in Microsoft Research, Redmond, WA.