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. 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.

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. 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Fortunately, Moore’s Law has not been repealed.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. 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

---

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.

Advanced Micro Devices multiple 45nm quad core die based on the Opteron processor, codenamed “Shanghai” (www.amd.com)
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, a forecast he subsequently reduced to every second year. 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. 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, 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-
Contributed articles have evolved from assembly language to structured objects (such as XML). The growing popularity of managed languages (such as .NET and the Java Class Library) also 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. 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.

Table 1 compares several key object-oriented complexity metrics between Windows 2003 and Vista, showing increased use of object-oriented features.

<table>
<thead>
<tr>
<th></th>
<th>Vista/Win 2003</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total Functions</strong></td>
<td>1.45</td>
</tr>
<tr>
<td><strong>Max Class Methods</strong></td>
<td>1.22</td>
</tr>
<tr>
<td><strong>Total Class Methods</strong></td>
<td>1.59</td>
</tr>
<tr>
<td><strong>Max Inheritance Depth</strong></td>
<td>1.33</td>
</tr>
<tr>
<td><strong>Total Inheritance Depth</strong></td>
<td>1.54</td>
</tr>
<tr>
<td><strong>Max Subclasses</strong></td>
<td>3.87</td>
</tr>
<tr>
<td><strong>Total Subclasses</strong></td>
<td>2.27</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Language</th>
<th>Debug Build</th>
<th>Optimized Build</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Working Set</td>
<td>Startup Bytes</td>
</tr>
<tr>
<td>C</td>
<td>1.424K</td>
<td>6,162</td>
</tr>
<tr>
<td>C++</td>
<td>6,756K</td>
<td>113,280</td>
</tr>
</tbody>
</table>

Table 3: .NET, or window systems; the 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. 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. 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
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.\(^4\) 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-
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.
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.7 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.” 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.