Spending Moore's Dividend

Microsoft Research Technical Report MSR-TR-2008-69

James Larus
5/2/2008

Spending Moore's Dividend

"Grove giveth and Gates taketh away." – anon.

James Larus
larus@microsoft.com
Microsoft Research

Summary: Thanks in large measure to the improvement in semiconductors predicted by Moore's Law, CPU performance has increased 40-50% per year over the past three decades. The advent of Multicore processors marks an end to sequential performance improvement and a radical shift to parallel programming. To understand the consequences of this change, it is worth looking back at where the thousands-fold increase in computer performance went and looking forward to how software might accommodate this abrupt shift in the underlying computing platform.

Introduction

The Intel 8080, introduced April 1974, comprised 4,500 transistors and ran at 2 MHz. The Intel Core 2 Duo, introduced July 2006, comprised 291 million transistors and ran at 2.93 GHz.[1] In a little over three decades, the number of transistors in a processor increased 64,467 times and clock speed 1,465 times. Hardware evolution naturally improved the performance of software. The Intel 80386DX (introduced October 1985) produced 0.5 SPEC CPU95, while the Intel Core 2 Duo produced 3108 SPECint 2000 base, an improvement of 661 times in approximately ten years.[2] Over Microsoft's history, regular, predictable improvements in the computing platform were the norm, culminating in thousand-fold increases in many measureable dimensions of a computer system. Most of these improvements are attributable to Moore's Law, the steady, 40% per year increase in transistors per unit area.

Over these decades, in which the personal computer and packaged software industries were born and matured, software development was facilitated by the comforting belief that every

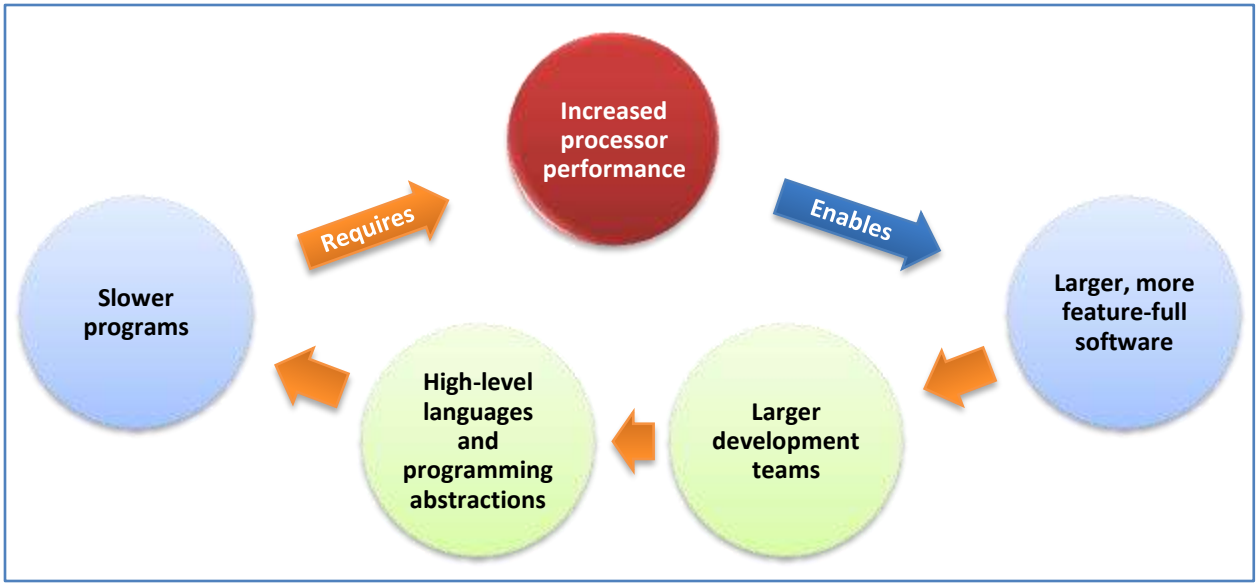


Figure 1. Cycle of innovation in computer industry.

generation of processors would run much faster than its predecessor. This assurance produced a cycle of innovation depicted in Figure 1. Faster processors enabled software vendors to add new features and function to software, which in turn required larger development teams. The challenge of building larger and more complex software, along with faster processors, increased demand for richer programming abstractions such as languages and libraries. These led to slower programs, which drove the demand for faster processors and closed the cycle.

The past few years mark the start of a historic transition from sequential to parallel computation in the processors used in most personal, server, and mobile computers. The introduction in 2004 of Multicore processors ends the remarkable 30-year period in which advances in semiconductor technology and computer architecture improved sequential processor performance at an annual rate of 40 – 50%. This improvement benefited all software, and it was a key factor driving the spread of software into the mainstream of modern life.

This era ended when practical limits on the power dissipation of a chip stopped the constant increases in clock speed and programs' lack of instruction-level parallelism diminished the benefit of increasingly complex processor architectures. The era did not end because Moore's Law was repealed. Semiconductor technology is still capable of doubling the transistors on a chip every two years. However, computer architects now use this flood of transistors to increase the number of independent processors on a chip, rather than making an individual processor run faster.

The goal of this paper is to try to study where the previous processor performance improvements were used by software, and to ask the obvious question of whether Multicore processors can satisfy the needs. In short, how did we spend Gordon Moore's dividend, and what can we do in the future?

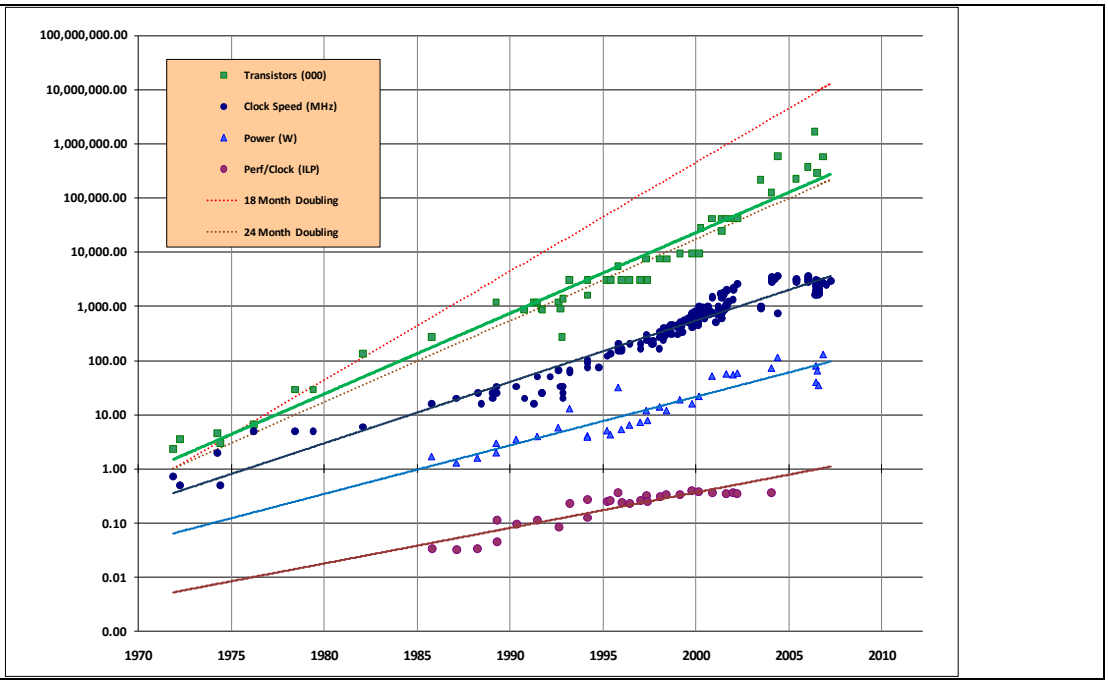


Figure 2. Improvement in Intel x86 processors. Data from Kunle Olukotum, Herb Sutter, and Intel.

Computer Performance

In 1965, Gordon Moore postulated that the number of transistors that could be fabricated on a semiconductor chip would increase at 40% per year.[3] It was a remarkable prediction that still holds today. Each generation of transistor is smaller and switches faster, thereby allowing

processor clock speed (and computer performance) to increase at a similar rate. Moreover, increased quantities of transistors enabled architects to improve processor designs by implementing sophisticated microarchitectures. For convenience, we will call the combination of these factors "*Moore's Dividend*." Figure 2 illustrates several aspect of 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 twenty-four months. Clock frequency, however, increased at a slightly slower rate. Computer architecture improvements came at an even slower rate: performance per clock cycle increased by only a factor of 14 over nearly 20 years.

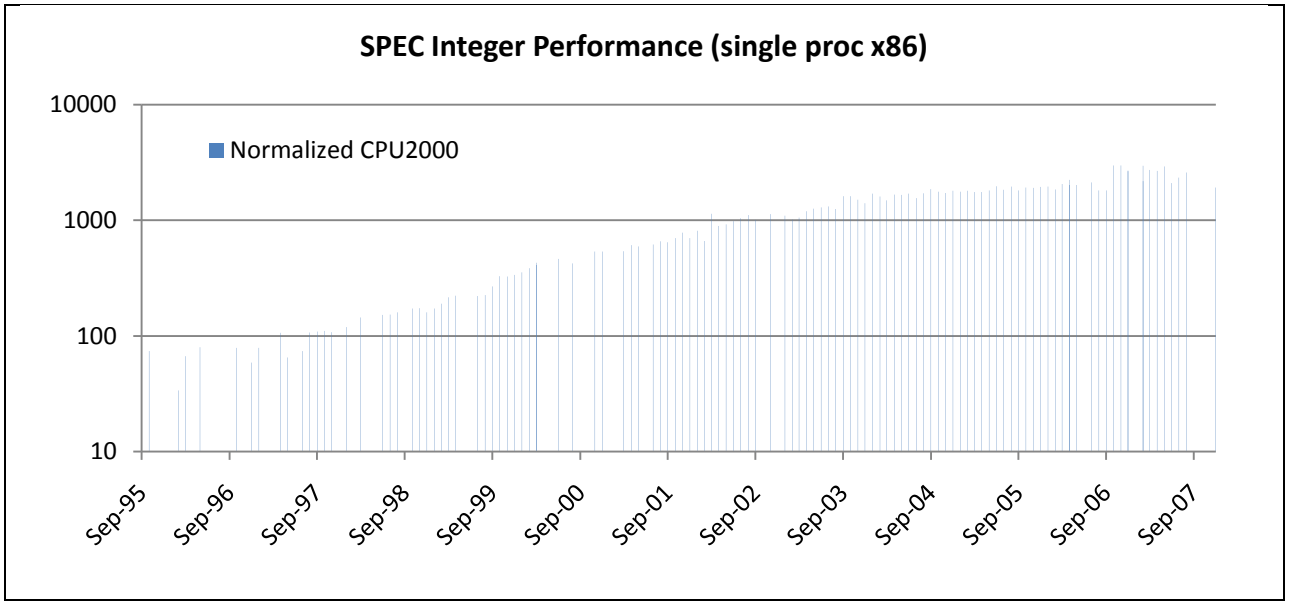


Figure 3. Performance improvement in single-processor (x86) SPEC CPU benchmarks. Data from www.spec.org.

Figure 3 shows the benefit of a decade of hardware evolution on software performance. The chart graphs the maximum SPEC CPU benchmark score reported each month for single-processor x86 systems (Intel and AMD). CINT95 and CINT2006 benchmark scores are normalized against CINT2000 using overlapping system configurations. Over the time period, integer benchmark performance improved 52x, almost the improvement of 57x obtained by doubling performance every two years.

Ekman et al. looked at a wider range of computers (including RISC processors) and found similar results.[4] Between 1985 and 1996, computer performance increased 50% annually, but the rate dropped to 41% between 1996 and 2004. Over the longer period, clock frequency increased 29% annually.

It is worth noting that some part of the overall performance improvement is attributable to compilers, though this contribution is regrettably small. Todd Proebsting found evidence in the SPEC Benchmarks for Proebsting's Law: compilers double program performance every 18 years (4% per annum).[5]

Moore's Law also drove improvement in memory chip capacity at a slightly faster rate than the more irregular processors (Figure 4).

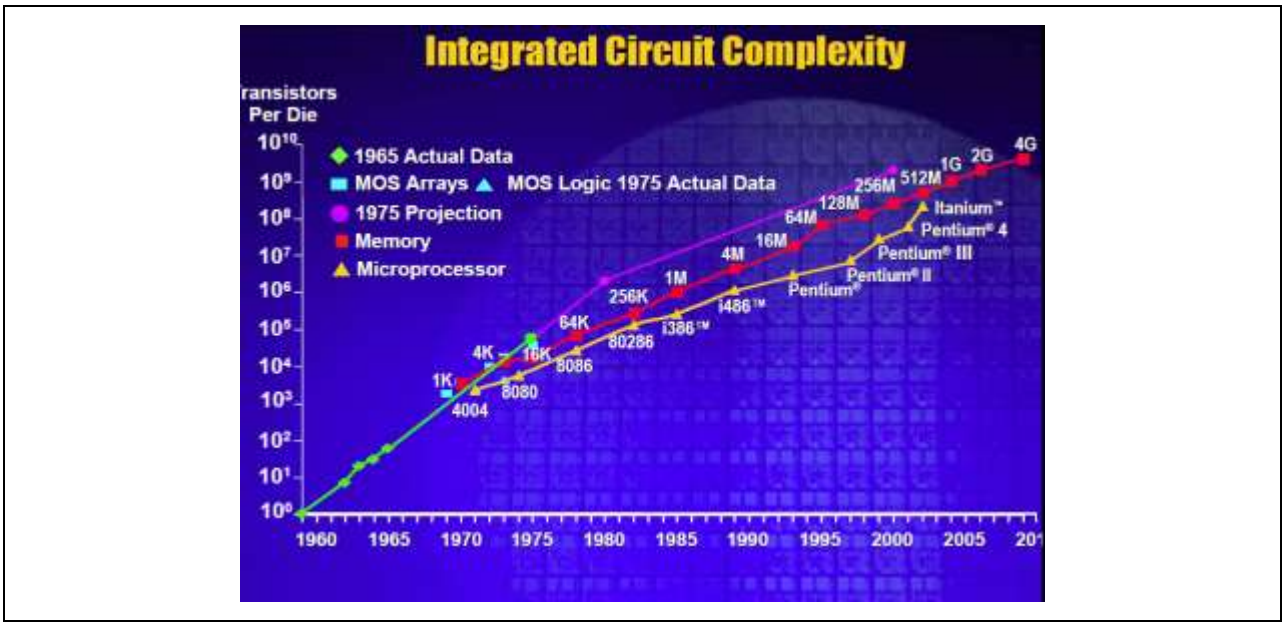


Figure 4. Capacity of DRAM chips. From Gordon Moore, "No Exponential is Forever," ISSCC, Feb. 2003.

Finally, though not directly connected to Moore's Law, but still very relevant to computing, disks have improved even faster than semiconductors. Figure 5 shows the improvement in areal ("bit") density in IBM (pre-2003) disk drives. From 1991 – 1997, areal density increased at 60% per year, and after 1997, it increased at 100% per year. Figure 6 shows that this rapid improvement drove down the cost of disk storage more aggressively than the cost of DRAM.

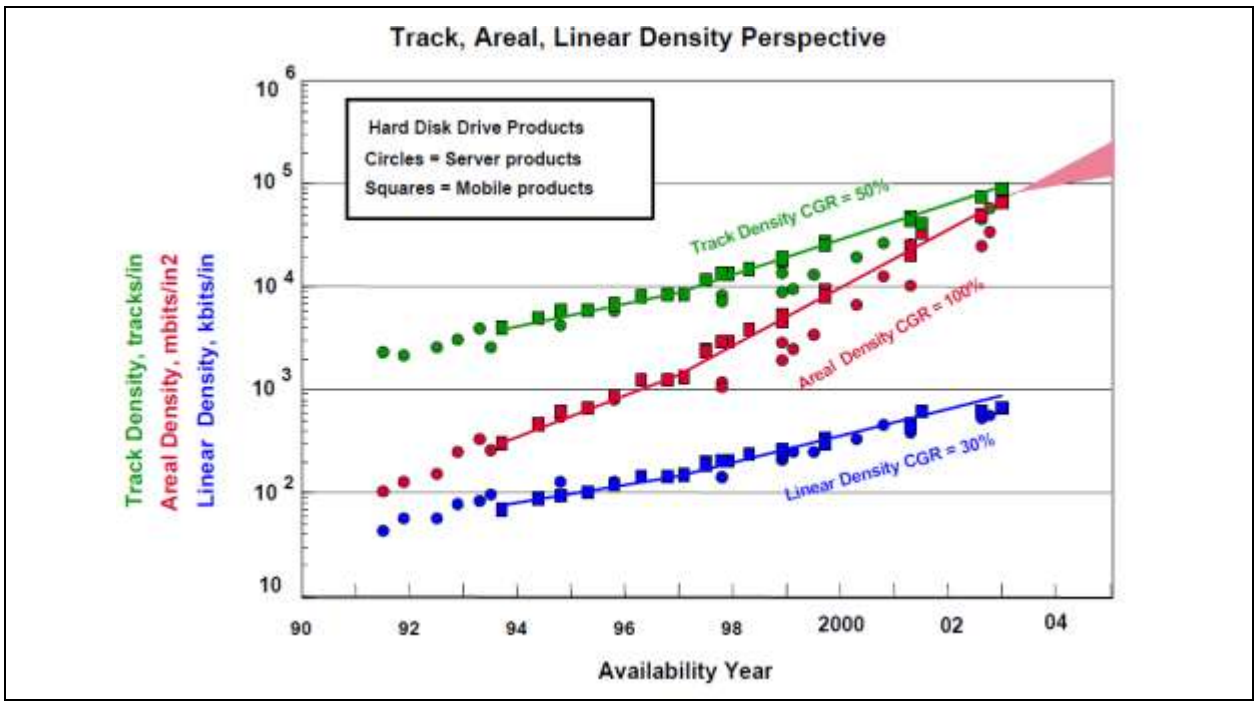


Figure 5. Improvements in IBM disk drives. (Hitachi GST Overview Charts, http://www.hitachigst.com/hdd/technolo/overview/storagetechchart.html)

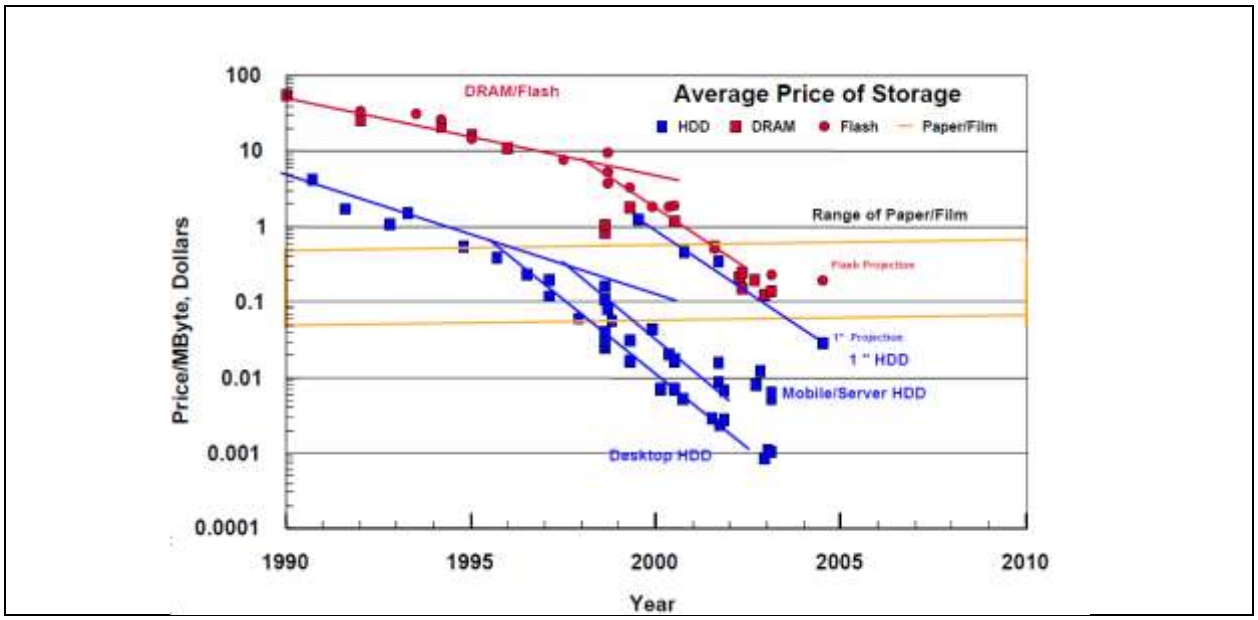


Figure 6. Cost of storage. (Hitachi GST Overview Charts, http://www.hitachigst.com/hdd/technolo/overview/storagetechchart.html)

Myhrvold's Laws and Microsoft Software

A widely held belief is that software grows in size at least as fast as processors increase in performance, memory in capacity, etc. An interesting counterpoint to Moore's Law is Wirth's Law: "software is slowing faster than hardware is accelerating."[6] Nathan Myhrvold memorably captured this wisdom with his four laws of software, starting from the premise that "software is a gas" (see sidebar).

For some Microsoft software, support for this belief depends on the metric used to measure the "volume" of software. Soon after Nathan published these "laws," the rate of growth of lines of

Nathan Myrvold's Laws of Software

(http://research.microsoft.com/acm97/nm/tsld026.htm)

1. 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)

2. Software grows until it becomes limited by Moore's Law

Initial growth is rapid - like gas expanding (like browser)
Eventually, limited by hardware (like NT)
Bring any processor to its knees, just before the new model is out

3. Software growth makes Moore's Law possible

That's why people buy new hardware - economic motivator
That's why chips get faster at same price, instead of cheaper
Will continue as long as there is opportunity for new software

4. It's impossible to have enough

New algorithms
New applications and new users
New notions of what is cool

code in Windows diverged dramatically from the Moore's Law curve (Figure 7). Initiatively, this makes sense: the size of a software system may increase rapidly in its early days, as basic functionality accrues rapidly, but exponential growth, such as the factor-of-four increase in lines of code (LoC) between Windows 3.1 and Windows 95 (2 years), is difficult to sustain without equivalent increases in headcount (or remarkable—and unprecedented—improvements in software productivity).

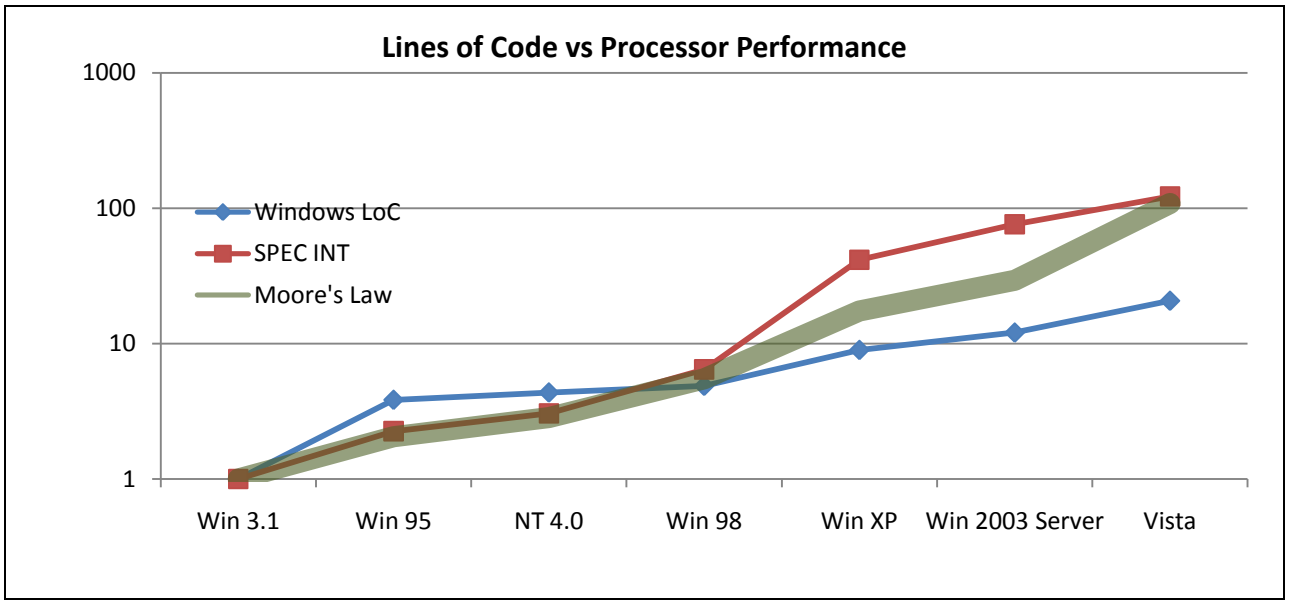


Figure 7. Windows source code size (LoC) and Intel Processor performance, July 1993 = 1.

Linux exhibits much the same pattern of growth (Figure 8).[7]

However, we can measure software volume in other ways, such as machine resources (processor performance, memory size, disk size, etc.). Figure 9 shows the minimum recommended system for different versions of Windows. With the exception of disk space (which increased faster than Moore's law), the minimum recommended Windows configurations grew at the same rate as the transistors, confirming a version of Myhrvold's Law.

It is worth noting that minimum recommended systems are far off the leading edge of the technology curve, so Microsoft developers had a strong incentive to push aggressively to increase the minimums. Marketing's countervailing desire was to keep the minimums low to maximize the potential market. It does not seem surprising that these forces balance out by increasing with Moore's Law, but the dynamics seem worthy of more study.

For this paper, the interesting fact is the divergence between Windows code size and system performance, which suggests that Moore's Dividend was spent on other factors than just code bloat. The gap between these two quantities, not just increased clock speed, made possible great improvements in system architecture, programming languages, and development style.

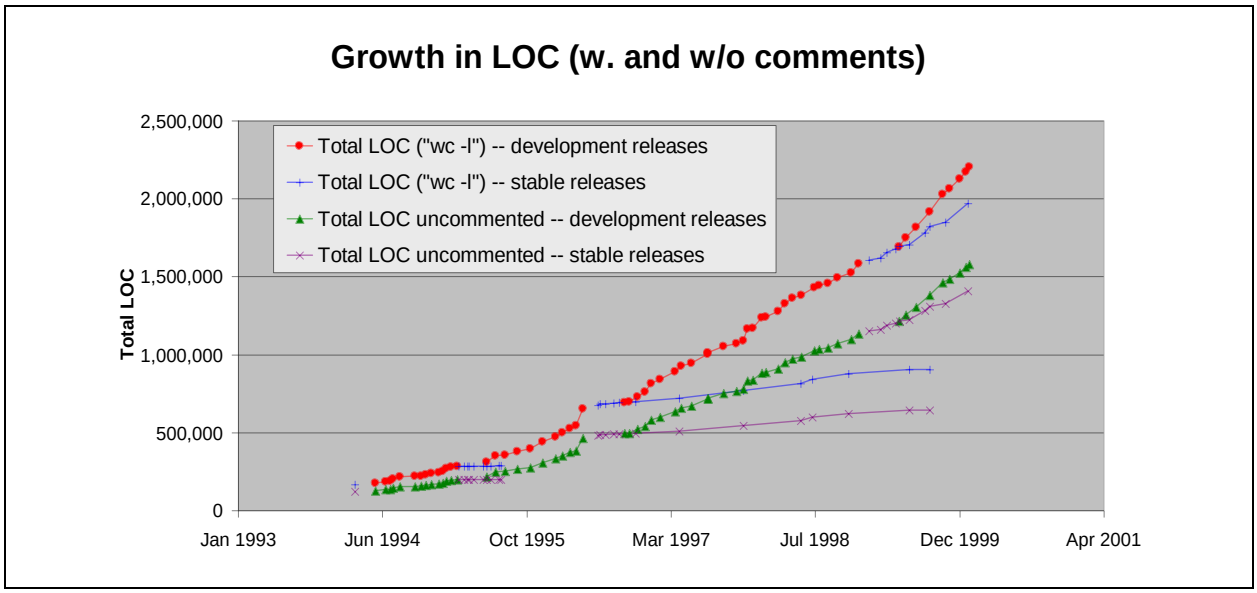


Figure 8. Growth of Linux (Godfrey 2000).

Where Did It Go?

Determining where Moore's Dividend was spent is difficult for a variety of reasons. The evolution discussed below occurred over a long period, and no one systematically measured shifting factors' effects on resource consumption. It is easy to compare released systems, but many aspects of a system or application change between releases, and without close investigation, it is difficult to attribute visible differences to changes in a particular factor.[8] The discussion presents some experimental hypotheses, which await further research to quantify their contributions more precisely.

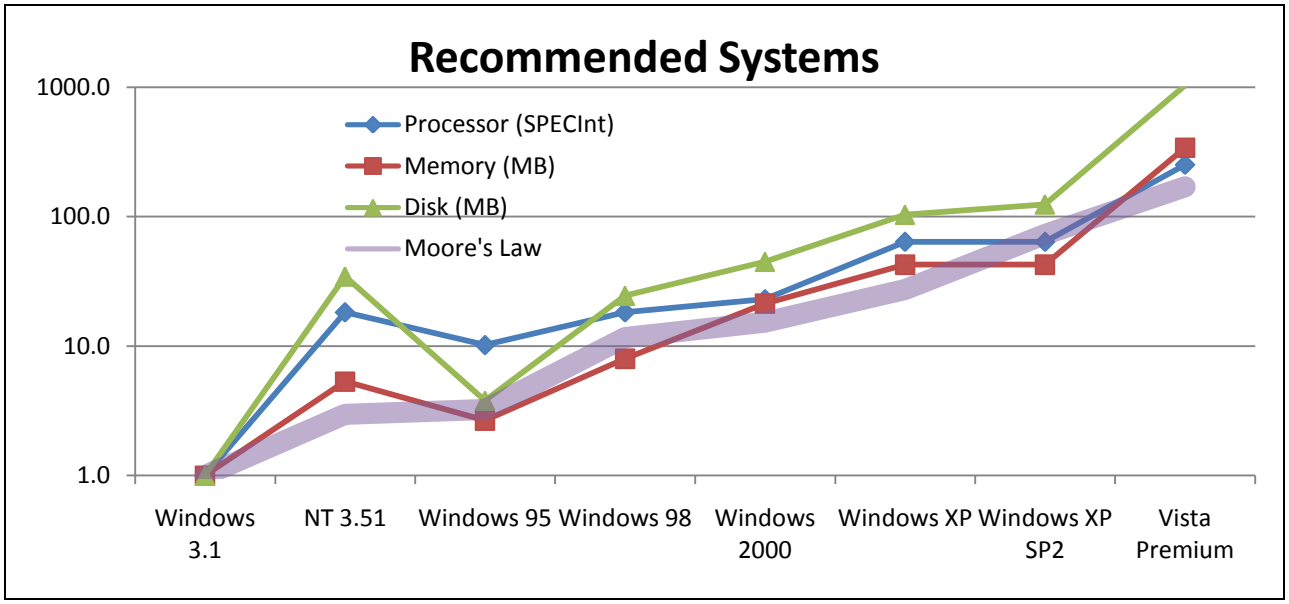


Figure 9. Recommended Windows configurations (high values from http://support.microsoft.com).

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, and their cumulative effect has been to steadily increase the amount of computation to perform a specific task, albeit in a more user-friendly or pleasing manner.

When I was in graduate school, the research community's aspirational goal was the "3M" workstation, which mimicked the personal workstations developed at Xerox PARC and promised 1 million instructions per second, 1 megabyte of memory, and a 1-million-pixel screen.[9] Today, such a computer would be inadequate even for a low-end cellphone (aside from the display).

To see how shifting expectations increase computation, it is helpful to focus on a specific component of a system. Far more popular and successful than the 3M workstation was the original IBM PC, whose monochromatic display adaptor (MDA) showed 25 lines of 80 characters and contained 4K of video memory.[10] A low-end computer today has SXGA (32-bit color, 1280x1024 resolution, 64MB of video memory). Graphics processing units (GPU) are far more capable than the MDA, in large measure because Moore's Law enabled them to become low-cost, high-performance vector processing units. Even so, performing an identical task (say, displaying text in a cmd shell window) on a modern computer entails far more computation. For one, display resolution shifted from 1 to 32 bits per pixel. In addition, the modern computer runs a more elaborate and capable window system, which entails many more levels of software abstraction and requires more computation to produce an equivalent visual effect. Of course, most of us prefer modern user interfaces to cmd shells, but overlapping windows, high quality fonts, graphics, etc. further increase the computation needed to manage the display. These two factors—new and improved functionality and the growth in software complexity to provide it—most certainly consumed a substantial fraction of Moore's Dividend. The exact proportion is difficult to quantify.

Improvements of this sort are pervasive throughout system and application software. I am writing this document in Microsoft Word, which performs background spelling and grammar checking, two features I have long since accepted as essential. When I use an editor or web control without these features, I cringe (and typically copy the prose into Word to check it). Features raise the expected level of functionality, and at the same time, raise the computational requirements.

As well as features, the data manipulated by a computer evolved over time, from simple ASCII text to larger, more structured objects, such as Word or Excel documents; compressed documents, such as JPEG images; and more recently to space-inefficient and computation-inefficient formats such as XML. The growing popularity of video introduces an exponentially larger object that is computationally very expensive to manipulate.

Another important, but difficult to quantify, change in our software is an increased concern with security. Attacks of various sorts have forced programmers to be more careful in writing low-level, pointer-manipulating code and increased scrutiny of input data. It is difficult to quantify security's effects on resource requirements. One hint is that array bounds and null pointer checks impose a time overhead of roughly 4.5% in the Singularity OS.[11]

Equally important, but difficult to measure, is the consequence of improved software engineering practices, such as Window's efforts to layer the software architecture and modularize the system to improve development and allow subsets.

Microsoft's strong support for legacy compatibility ensures that the tide of resource requirements rises, and never recedes. Features added to software are rarely removed, since it is difficult to ensure that no customers use them.[12] Office reputedly studied use of their applications

and found that a given person may only use 10% of an application's features, but taken together, all Office functionality was used.[13]

	Relative to WinXP						
	Size Increase		New Software		Legacy Code		
	Files	Lines	New Files	Added or Changed Lines	Untouched Files	Edited Files	Original Lines
Win 2k3	1.43	1.42	1.11	1.13	0.73	0.93	0.78
Vista	1.80	1.46	1.07	1.03	0.80	1.00	0.94

Table 1. Legacy code in Windows, from Murphy and Nagappan 2006. Numbers are change, relative to Windows XP.

As mentioned above, systems increase in size over time. Brendan Murphy and Nachi Nagappan's report on Vista Development offers a quantitative perspective on code change over time.[14] Table 1 shows the relative growth in size between three releases of Windows. In addition, it shows that very little of the code is left unchanged between releases, as bug fixes and improved functionality propagates through the system.

These phenomena are not specific to Microsoft software. Figure 10 shows the growth of the major subsystems of the Linux kernel in the 1990s.[15] The sharpest growth was in drivers, the component of an OS kernel that increases the functionality of the system by allowing it to control more hardware devices. The other subsystems in the kernel clearly evolved and matured over the interval, but the changes were not apparent to most users as Linux kept a stable, Unix-like interface to its kernel. The noticeable improvement was in the range of systems that Linux could run on and the number of hardware devices that it could use.

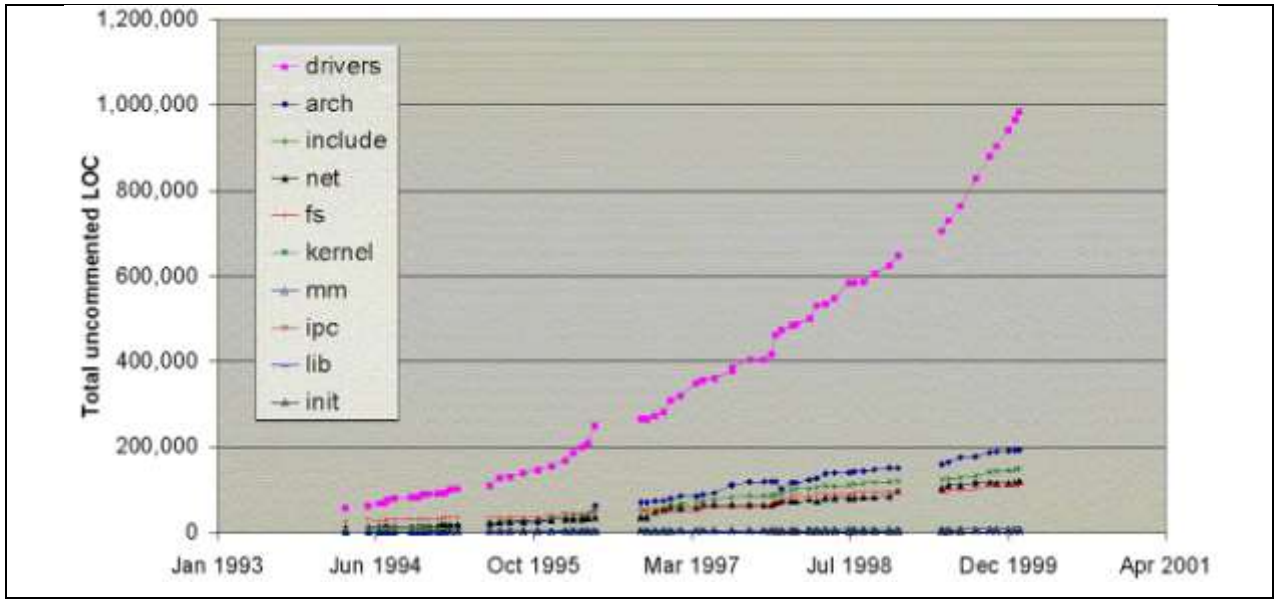


Figure 10. Size of major subsystem of Linux kernel. From Godfrey and Tu 2000.

Programming Changes

Another major change over the past 30 years is the evolution of programming from low-level, assembly language and C programming to increased use of higher-level languages that provide richer programming abstractions. The first step was a move to object-oriented languages, specifically C++, which introduced mechanisms, such as virtual method dispatch, whose implementation and cost were not always transparent to developers. Equally costly were abstraction mechanisms, such as templates, that made possible rich libraries such as the C++ Standard Template Library (STL). These language and run-time mechanisms, some with non-trivial and opaque implementations, improved programming practice by provided new abstraction mechanisms that improved modularity and information hiding and increased code reuse. In turn, these improvements enabled the construction of the larger and more complex software systems required to provide new functionality.

Data in Murphy and Nagappan's report can quantify the shift to an object-oriented style of programming in Windows. Table 2 compares object-oriented complexity metrics between Windows 2003 and Vista. The metrics show an increased use of object-oriented features between the two systems. For example, the mean number of classes per binary had a 59% increase, and the mean number of subclasses per binary increased 127%.

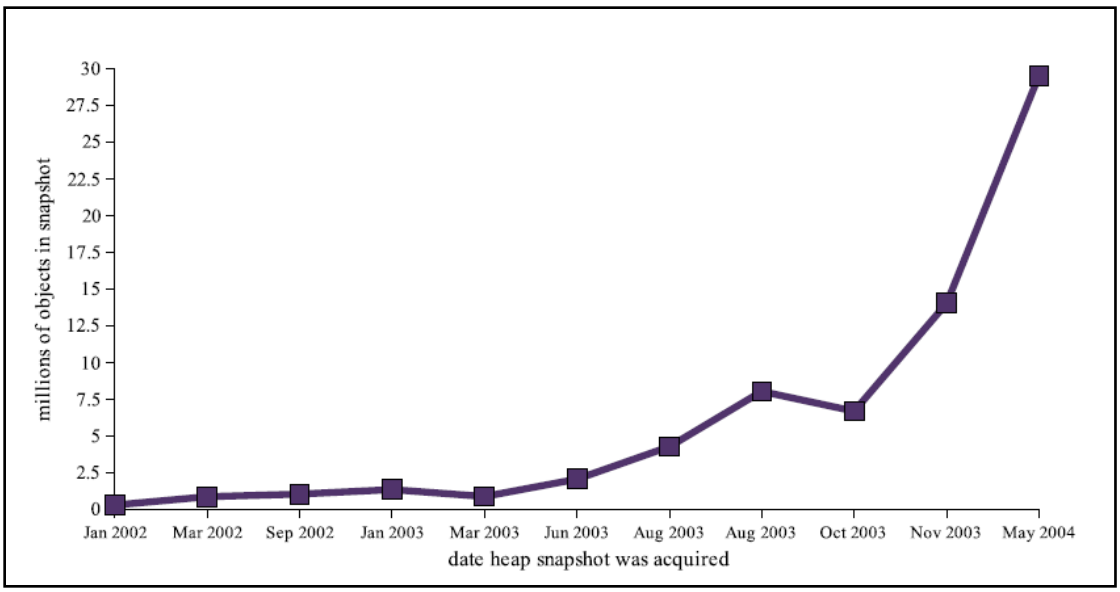
Kejariwal et al. compared the performance of SPEC CPU2000 and CPU2006 and attributed the lower performance of the newer benchmark suite to an increase in complexity and size due to the inclusion of six new C++ benchmarks and enhancements to existing programs.[16] For example, the growing size of the programs common to the suites, along with their increased use of memory pointer, frustrated compiler optimization, decreasing IPC (instructions per cycle) from 1.65 to 1.29 and increasing the L1D cache miss rate from 10.2 to 12.9.

	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 Sub Classes	3.87
Total Sub Classes	2.27

Table 2. Object-oriented complexity metrics (per binary). From Murphy and Nagappan.

Safe, modern languages, such as C# and Java, further increases the level of programming abstraction by introducing garbage collection, richer class libraries (.NET and the Java Class Library), just-in-time compilation, and powerful reflection facilities. All of these features provide powerful programming abstractions for developers, but also consume memory and processor resources in non-trivial ways and obscure the link between a developer's intuition and program behavior.

Mitchell et al. analyzed the process of converting a date in SOAP format to a Java Date object in IBM's Trade benchmark, a sample business applications built on IBM Websphere middleware. The conversion entailed 268 method calls and allocation of 70 objects.[17] Another paper reported the data in Figure 11, illustrating a rapid increase in the number of objects in the memory of a variety of large-scale server applications.[18] Jann et al. analyzed this benchmark on consecutive implementation of IBM's POWER architecture, and noted "Modern eCommerce applications are



increasingly built out of easy-to-program, generalized but non-optimized software components, resulting in substantive stress on the memory and storage subsystems of the computer."[19]

Figure 11 Objects in memory in "variety of large-scale applications." (From Mitchell 2006.)

As a simple, experiment, I wrote Hello World in C and C# and compiled both versions on a x86 running Vista Enterprise, using Visual Studio 2008. Table 3 shows the machine resources for both versions. The C# program had a working set 4.7 – 5.2 times larger. This is not a criticism of C# or .NET – the overhead comes along with a system that provides a much richer set of functionality that makes programming faster and less error-prone. Moreover, the differences between the two environments are far less than the performance difference between the computers of the 1970s and 1980s, when C originated, and today.

	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	87,662

Table 3. Hello world benchmark.

Increased abstraction in programming languages has two consequences for program performance. The first is that a new language mechanism can impose a resource cost, even where it is not used. A well-known example, language reflection requires the run-time system to maintain a large amount of metadata in case that feature is invoked. The second consequence is that higher-level languages hide details of machine performance beneath a more abstract programming model. Increased abstraction both makes developers less aware of performance concerns and less able to understand and correct problems when they arise. Consider .NET. Few developers have an intuitive model of how code should perform on that platform, in part because of the complexity and large number of features provided by the system. An MSDN article presents this (partial) list of factors that influence .NET performance:[20]

- Memory management
    - GC.Collect
    - Finalizers
    - Unmanaged resources
    - IO Buffers

- Looping and recursion
    - Repeated property access
    - Recursion
    - foreach
    - String concatenation
    - StringBuilder
    - String comparisons
- Exception handling
    - Catching and not handling exceptions
    - Controlling application logic with exceptions
    - Finally blocks
    - Exceptions in loops
    - Rethrowing exceptions
- Arrays
    - …
- Collections
    - …
- Locking and synchronization
    - …
- Threading
    - …
- Asynchronous Processing
    - …
- Serialization
- …

Decreased Programmer Focus

Huge increases in machine resources have made programmers more complacent about performance and less aware and concerned about resource consumption in their code. Thirty years ago, Bill Gates changed the prompt in Altair Basic from "READY" to "OK" to save 5 bytes of memory.[21] Today, it is inconceivable that developers would be aware of this level of detail of their program, let alone concerned about it, and rightly so, since a change of this magnitude is today unnoticeable.

More significant, however, is a change in developers' mindset, to be less aware of the resource requirements of the code they write. There are several good reasons for this change:

- Increased computer resources mean that fewer programs are pushing the bounds of a computer's capacity or performance, and hence many more programs will run on existing computers without extensive tuning. Knuth's widely known dictum that "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 does not matter if it consumes twice the resources it might require after further tuning. However reasonable, this practice ensures that many programs will execute at or near machine capacity, and consequently guarantees that the Moore's Dividend will be fully spent by every new release by developers happy to let chip designers assume the burden of good performance.
- Another important change is that today large teams of developers write software. The performance of a single developer's contribution is often difficult to understand or improve in isolation (i.e. performance is not modular). As the systems become more complex,

incorporate more feedback mechanisms, and run on less predictable hardware, it is increasingly difficult for an individual developer to understand the performance consequences of his or her decisions. A problem that is everyone's responsibility is no one's responsibility.

- The performance of computers is far more difficult to understand or predict. It used to suffice to count instructions 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 program behavior and machine architecture to predict program performance. Multicore processors, unfortunately, are likely to continue this trend.
- Programs written in high-level languages depend on compilers to achieve good performance. Compilers generate remarkable good code on the average, but they can perform badly for algorithms that make non-standard use of hardware features or that use hardware that the compiler cannot compile for effectively.

None of these considerations is a criticism of current development practices. There is no way we could produce today's systems using the artisan, handcrafted practices that were possible (necessary) on machines with 4K of memory.

Shifting Source of Bottlenecks

Most of this paper focused on the CPU, since the shift to Multicore affects that component. Historically, as CPUs improved rapidly, other components lagged and became bottlenecks. The two prominent examples are storage, both DRAM and disk. DRAM performance (access time) has increased at roughly 7% per year, which means the gap between a processor and its memory has increased 30-40% per year. To alleviate this mismatch in speed, processors have increased the size and number of caches and incorporated latency-tolerant micro-architectures. Nevertheless, the poor memory locality of most programs ensures they run far below a processor's peak potential. Disks offer a similar story. Capacity grew even faster than Moore's Law (Figure 5), but access table decreased at a far slower rate (Figure 12). File system access is now a major responsiveness issue. Storing data on a server exacerbates the problem by introduces networking and server queuing delays as well.

These bottlenecks have several consequences for processor performance. On one hand, these bottlenecks are largely responsible for the perception that computers are "fast enough" to execute common programs. In some sense, they are, if memory and disk accesses limit the performance of software running on even slow or moderate speed computers. However, engineering around these bottlenecks consumes computer resources, both memory and processor cycles, to manage data cache and redesign code to tolerate unpredictable latencies.

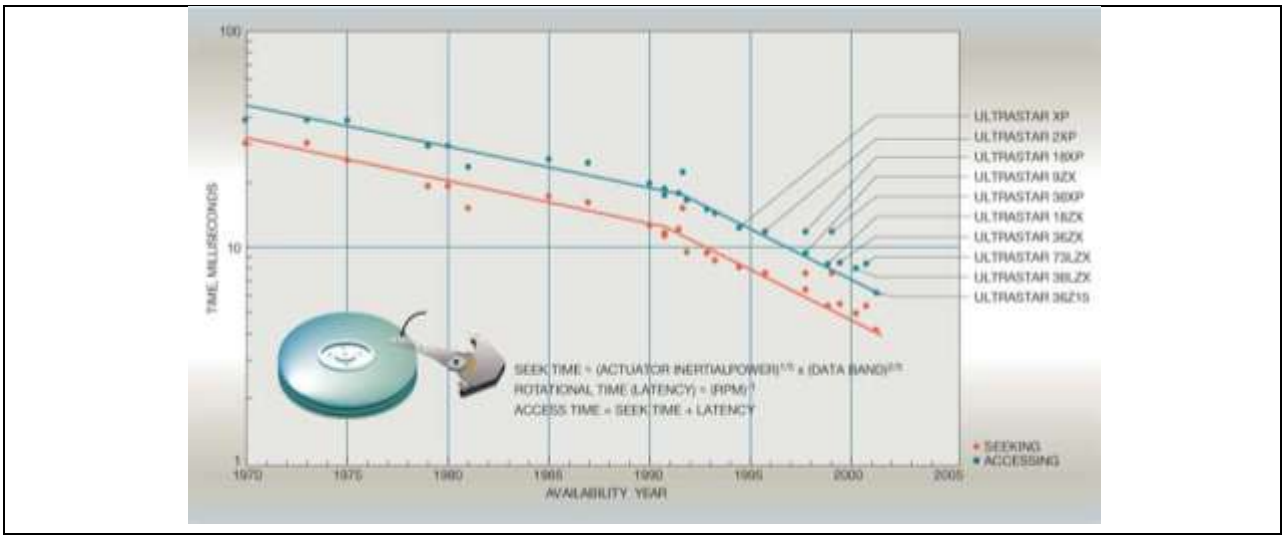


Figure 12. Access time improvement of IBM drives. From E. Grochowski and R. D. Halem, "Technological impact of magnetic hard disk drives on storage systems," http://www.research.ibm.com/journal/sj/422/grochowski.html.

The imminent changes in mass storage (flash and phase change memory) will greatly reduce disk access time and mitigate a major bottleneck in systems. Eliminating this bottleneck is important to being able to use Multicore processors effectively, as many parallel problems target large data sets that reside on disk.

Multicore and the Future

No doubt, every reader could think of other places 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.

Software Evolution

First, consider the normal process of software evolution, the extension and enhancement of existing, sequential system and application code. This category of software excludes systems and applications already running on parallel computers (e.g. DBs, web servers, scientific applications, games, etc.), which presumably will continue to perform well on increasingly integrated Multicore processors. The normal process of software evolution might exploit parallel computers in several possible ways:

The best case is when a new program version adds functionality that uses a parallel algorithm to solve a computationally or data-intensive task. This problem would have to have been solvable on a conventional processor, if sequential performance had continued to increase. Multicore is not a magic performance elixir, but rather just a different way in which the exponential increase in transistors can be turned into greater computer performance. In particular, Multicore does not change the rate of performance improvements, aside from one-time architectural shifts, such as replacing a complex processor by many simple cores.

Designing and developing a parallel algorithm is a considerable challenge, but many problems, such as video processing, natural language and speech recognition, linear and non-linear optimization, and machine learning, are computationally intensive. If the challenge precluding widespread usage of these algorithms is computational, rather than algorithmic, and if parallel algorithms exist or can be developed, then Multicore processor could enable compelling new

application scenarios. Adding major new functionality is a clear way in which existing applications can exploit Multicore processors.

Functionality of this type will have a different character than the common, more evolutionary features typically added to a new version of a software product. A feature only requires parallelism if its computation consumes an entire processor (or more) for a significant amount of time, which was obviously impractical for most features in existing software. Small, incremental improvements to functionality are unlikely to require more than one processor for a short amount of time, a model that does not scale as the number of cores increases or provide a compelling rationale to make the computation parallel. A parallel "killer app" requires big, bold steps forward.

An alternative approach is to redesign a sequential application into a loosely coupled or asynchronous system, in which multiple computations run on separate processors. This approach uses parallelism to improve the architecture or responsiveness of a system, rather than its performance. For example, it is natural to separate system monitoring and introspection features from program logic. Running these features on a separate processor reduces perturbation of the main-line computation, but entails synchronization and communication among processors. Alternatively, extra processors can perform speculative computations to decrease response time. A computation could be started in the background before its value is required, or several alternatives could be started together in a race to produce an answer. The amount of available parallelism is unlikely to grow with Moore's Law, but since many systems run multiple applications, "space sharing" of small clusters of processors can produce a more responsive system, at the cost of inefficiently using the abundant computing resources.

New or existing functionality that does not fit these two patterns will not benefit from Multicore, and instead it will remain constrained by the static performance envelope of a single processor. The performance of a processor may continue to improve at a significantly slower rate (optimistic estimates range around 10-15% per year). On many chips (e.g., Intel's Larrabee), processors will be significantly slower, as chip vendors simplify individual cores to lower power consumption and integrate more processors on a chip.

For many applications, most functionality is likely to fall in this category. For them, it will be possible to add or evolve features only by eliminating old features or reducing their resource consumption. A paradoxical consequence of Multicore is that sequential performance tuning and code-restructuring tools are likely to become increasingly important. Another likely consequence is that software vendors will become more aggressive in deprecating old or redundant features, to make space for new code.

The regular increase in Multicore parallelism poses an additional challenge to software evolution. Kathy Yelick, a researcher in HPC programming languages and tools, noted that the experience of the HPC community was that each decimal order of magnitude increase in available processors required a major redesign and rewriting of parallel applications.[22] Multicore processors are likely to come into widespread use at the cusp of the first such change ($8 \rightarrow 16$), and the next one ($64 \rightarrow 128$) is only three processor generations (6 years) later. This observation is, of course, only relevant to applications that use scalable algorithms and face scalable problems that require large numbers of processors. A majority of parallel applications are likely to need only a fraction of future Multicore machines and so remain oblivious to this challenge.

Software Development

Parallelism will force major changes in software development. Moore's Dividend enabled the shift to higher-level languages and libraries. This trend is unlikely to end because of its

beneficial effects on security, reliability and program productivity, but its emphasis will inevitably shift to parallelism.

Parallelism can enable new implementations of existing languages and features. For example, concurrent garbage collectors reduce the pause time of computational threads, thereby enabling the use of safe, modern languages in a wider range of applications, such as real-time programming. On the other hand, a recent trend is increasing interest in dynamic languages, such as Python, Perl, or Javascript. None of these languages is parallel, nor does parallelism offer an obvious path to reducing the large overhead of this type of language.

More relevant is a renewed interest in functional languages, such as Haskell and F#, that reduce the use of imperative programming constructs, which require synchronization in parallel programs. These languages also appear to offer a smoother evolutionary path to incorporate parallel programming constructs. For example, Microsoft Research's Haskell-TM system demonstrated simple, natural solutions to difficult problems, such as weak atomicity, IO, and transaction coordination, none of which have good alternatives in transactional memory systems for imperative languages.[23] Functional languages may not be appropriate for all types of programming, but they have a wider range than commonly perceived. For example, Erlang is a functional language used to construct real-time systems for commercial phone switches and internet routers.[24]

Another trend is the design of domain-specific languages that provide an implicitly parallel programming model, in which a developer is not aware of the individual processors, but instead writes in a language whose semantics is appropriate for parallel evaluation. For example, there is an interesting project at UC Berkeley to replace the Javascript language with a parallel alternative (Flapjack), as part of a project to build a parallel browser.[25] The difficulty here is that domain-specific languages are, by definition, not general purpose programming languages, and good languages are costly and difficult to implement.

Many parallel programming constructs and languages have been proposed, but none is in widespread use. The most mature proposals support data parallel programming, an important component of some numeric and image processing, but far from a universal parallel programming model.

The Singularity experience in MSR suggests that a radical change is an opportunity to rethink the time-tested software stack. A systematic redesign can make new ideas (e.g., safe languages) more palatable to developers, while at the same time providing an opportunity to solve old problems.[26] Experience with Transactional Memory drives home the same need for systematic change through the software stack. Transactions at first appeared to offer a clean and effective parallel programming construct, but the complications of integrating them into existing languages and environments raised the possibility that the cure is worse than the disease. Grafting parallel constructs onto existing parallel languages may satisfy the need to write code for Multicore processors, without providing an effective means of exploiting parallelism.

Parallel Software

The more interesting category of applications and systems are those designed to take advantage of parallelism. Currently, the two prominent areas that use parallel computation are servers and high-performance computing. These two areas provide different, but important lessons. Servers have long been the commercially successful parallel systems, as their workload consists of mostly independent requests that are "embarrassingly parallel," in that they require little or no coordination and share data through a coarse-grain transaction mechanism such as a DB. As such, it is relatively easy to build a web application, since the programming model and

libraries are high-level abstractions that express the processing for each request as a sequential computation. Building an application (web site) that scales well is still an art, because scale comes from replicating machines, and coordinating, communications, and managing across machine boundaries breaks the sequential abstraction.

High-performance computing followed a different path. This field uses parallel hardware because there is no alternative to obtain sufficient performance, not because scientific and technical computations are well suited to parallel solution. Parallel hardware is a tool to solve a problem. The programming models (MPI and OpenMP) are low-level abstractions that are highly error-prone and difficult to use effectively. More recently, game programming emerged as another realm of high-performance computing, with the same characteristic of talented, highly motivated programmers spending tremendous effort to achieve the last bit of performance out of difficult hardware.[27]

If parallel programming is to become mainstream, as Multicore supporters assume, then programming must be closer to servers than high-performance computing. The problems should be inherently parallel, with sufficient computation and little communication or synchronization among the processors. In addition, software and tools vendors must provide a rich collection of programming abstractions and libraries that hide the reality of the underlying parallel implementation. A popular example, Google's map-reduce utilizes a simple, well-known programming model for initiating and combining independent tasks, but more importantly it hides the complexity of running these tasks across a huge collection of computers.[28]

An alternative paradigm for parallel computing, Software plus Services revisits the time sharing, client-server, or thin client model by executing some or all parts of an application program on a server. Companies such as Salesforce.com argue for the operational and practical advantages of this model of computing.[29] Equally compelling is that this style of computing, like servers in general, is embarrassingly parallel and directly benefits from Moore's Dividend. Each application instance running on a server is essentially independent and executes naturally on a parallel computer. Moreover, Moore's Dividend accrues directly to the service provider. Each new generation of Multicore processors halves the number of computers need to serve a fixed workload or provides the headroom to add features or handle increasing workloads. Despite the challenges of this software paradigm (finding a scalable business model, handling limited or inconsistent connectivity, and providing off-line operation, etc.), this model of computation is likely to become increasingly popular.

Conclusion

The key message of this paper is that the Moore's Dividend of improved sequential performance have been widely spent throughout programming languages, models, architectures, and development practices, up through the user-visible behavior and functionality of software. Currently, it is difficult to see how parallelism could meet all of these demands, even if Microsoft and other software vendors are able to identify and build new functionality uniquely enabled by parallelism. Without a seismic shift to a new parallel programming model, languages, and tools and large effort to reeducate developers, future software evolution is likely to rely on developers' ability to "do more with less."

Acknowledgements

Many thanks to Al Aho, David Callahan, Mark Hill, and Scott Wadsworth for helpful comments on earlier drafts of this paper and to Nachi Nagappan for assistance with the internal Microsoft data.

Notes

[1] Processor metrics from http://download.intel.com/pressroom/kits/IntelProcessorHistory.pdf

[2] Benchmarks from the 8080 era look trivial today and tell little about modern processor performance, so a realistic comparison over the decades requires a more capable microprocessor than the 8080. The earliest SPEC Benchmarks results for Intel processors are from the 386. The revision of SPEC benchmark every few years complicates direct comparison. Moreover, the ratio of results between different suites change over time as processors and compilers improve. For this computation, I picked a point, late in 1999, when the Dell Precision WorkStation 420 (733 MHz PIII) produced 336 SPEC CInt2000 and 35.7 SPEC CPU95, a ratio of 9.4.

[3] G. E. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, pp. 56-59, April 1965.

[4] M. Ekman, F. Warg, and J. Nilsson, "An In-depth Look at Computer Performance Growth," *ACM SIGARCH Computer Architecture News*, vol. 33, pp. 144-147, March 2005.

[5] http://research.microsoft.com/~toddpro/papers/law.htm

[6] P. Ross, "Engineering: 5 Commandments," *IEEE Spectrum*, vol. 40, no. 12, pp. 30-35, December 2003. http://dx.doi.org/10.1109/MSPEC.2003.1249976

[7] M. W. Godfrey, D. Svetinovic, Q. Tu, "Evolution, Growth, and Cloning in Linux: A Case Study," University of Waterlo, Powerpoint. http://plg.uwaterloo.ca/~migod/papers/2000/cascon00-linuxcloning.ppt

[8] Widely publicized comparisons, such as OfficeBench (http://www.xpnet.com/iworldtest), provide some longitudinal data, but lack transparency about the details of the benchmark and the benchmarked systems, which makes it difficult to interpret, let alone reproduce the results.

[9] G. Bell, "Three Rivers PERQ: The Pioneer Gets the Arrows," http://research.microsoft.com/~gbell/High_Tech_Ventures/00000334.htm

[10] http://en.wikipedia.org/wiki/Monochrome_Display_Adapter

[11] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. R. Larus, "Deconstructing Process Isolation," in *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness,* San Jose, CA, 2006, pp. 1-10.

[12] As an example of what happens when a feature is deprecated, consider: J. Wilcox, "SP3 Downgrades Office 2003," *eWeek Microsoft Watch,* January 4, 2008. http://www.microsoft-watch.com/content/business_applications/sp3_downgrades_office_2003.html

[13] J. Harris, "Inside Deep Thought (Why the UI, Part 6)" discusses the Office SQM data used to assess usage of Office features. http://blogs.msdn.com/jensenh/archive/2005/10/31/487247.aspx

[14] B. Murphy, N. Nagappan, "Characterizing Vista Development," Microsoft Research, December 15, 2006.

[15] W. Scacchi, "Understanding Open Source Software Evolution," in *Software Evolution and Feedback*, N. H. Madhavji, M. M. Lehman, J. F. Ramil, and D. Perry, Eds. New York, NY: John Wiley and Sons, 2004.

[16] A. Kejariwal, G. F. Hoflehner, D. Desai, D. M. Lavery, A. Nicolau, and A. V. Veidenbaum, "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: ACM, 2007, pp. 361-362. http://doi.acm.org/10.1145/1254882.1254930

[17] N. Mitchell, G. Sevitsky, and H. Srinivasan, "The Diary of a Datum: An Approach to Analyzing Runtime Complexity in Framework-Based Applications," in *Workshop on Library-Centric Software Design*, San Diego, CA, 2005, pp. 85-90. http://domino.research.ibm.com/comm/research_people.nsf/pages/nickmitchell.pubs.html/$FILE/lcsd2005.pdf

[18] N. Mitchell, "The Runtime Structure of Object Ownership " in *Proceedings of the 20th European Conference on Object-Oriented Programming*: Springer, 2006, pp. 74-98. http://www.springerlink.com/content/tq2j22377r414143/

[19] J. Jann, R. S. Burugula, N. Dubey, and P. Pattnaik, "End-to-end Performance of Commercial Applications in the Face of Changing Hardware," *ACM SIGOPS Operating Systems Review*, vol. 42, pp. 13-20, January 2008. http://doi.acm.org/10.1145/1341312.1341317

[20] http://msdn2.microsoft.com/en-us/library/ms998574.aspx

[21] B. Gates, email, April 10, 2008.

[22] K. Yelick, discussion at Microsoft Research "AltTAB" on parallelism July 19, 2006.

[23] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable Memory Transactions," in Proceedings of the *Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Chicago, IL: ACM, 2005, pp. 48-60. http://doi.acm.org/10.1145/1065944.1065952

[24] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," in Department of Microelectronics and Information Technology. PhD, Stockholm, Sweden: The Royal Institute of Technology, 2003.

[25] Berkeley Parallel Browser, http://parallelbrowser.blogspot.com/2007/09/hello-world.html

[26] G. Hunt and J. Larus, "Singularity: Rethinking the Software Stack," *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 37-49, April 2007. http://doi.acm.org/10.1145/1243418.1243424

[27] T. Sweeney, "The Next Mainstream Programming Language: A Game Developer's Perspective," presentation at the *33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* Charleston, SC: ACM, 2006. http://www.cs.princeton.edu/~dpw/popl/06/Tim-POPL.ppt

[28] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, pp. 107-113, January 2008. http://doi.acm.org/10.1145/1327452.1327492

[29] Nicholas Carr argues that Software as a Service will replace desktop computing and company-managed data centers through an interesting analogy to the centralization of electric production and distribution in the early 1900s. N. Carr, *The Big Switch: Rewiring the World, From Edison to Google*. W.W. Norton, 2008. The latter two-thirds of the book, a bleakly dystopian vision of the consequence of this shift to web-centric software, is also well worth reading.