Brief History of Computer Architecture Evolution and Future Trends

R. D. Groves
IBM, Austin, Texas

Abstract

Computer system architecture has been, and always will be, significantly influenced by the underlying trends and capabilities of hardware and software technologies. The transition from electromechanical relays to vacuum tubes to transistors to integrated circuits has driven fundamentally different trade-offs in the architecture of computer systems. Additional advances in software, which includes the transition of the predominant approach to programming from machine language to assembly language to high-level procedural language to object-oriented language, have also resulted in new capabilities and design points. The impact of these technologies on computer system architectures past, present, and future will be explored and projected.

General

The key hardware technologies that affect computer architectures are those that determine the density and speed of digital switches and the density and access time of digital storage. Figure 1 shows the trend in increasing capacity and performance for digital switching technologies. The important trend to notice is that capacity improves at a more rapid rate than transistor speed. Figures 2 and 3 show the trends in capacity and access time for two important digital storage technologies: dynamic random access memories and magnetic disk storage. Here again, the important trend reflected is that capacity is improving at a faster rate than time taken to access the data stored. These basic trends have been true throughout the history of computer systems and are projected to continue through the foreseeable future in the absence of fundamentally new technological approaches.

Despite these underlying trends, the performance of computer systems has increased at a rate which exceeds the rate of improvement in transistor speed and digital storage access speeds. To achieve this, computer systems have been required to take approaches that improve performance by exploiting both technological density and speed. Since these basic trends are projected to continue, future system designs can be expected to leverage both density and speed to achieve additional system performance.

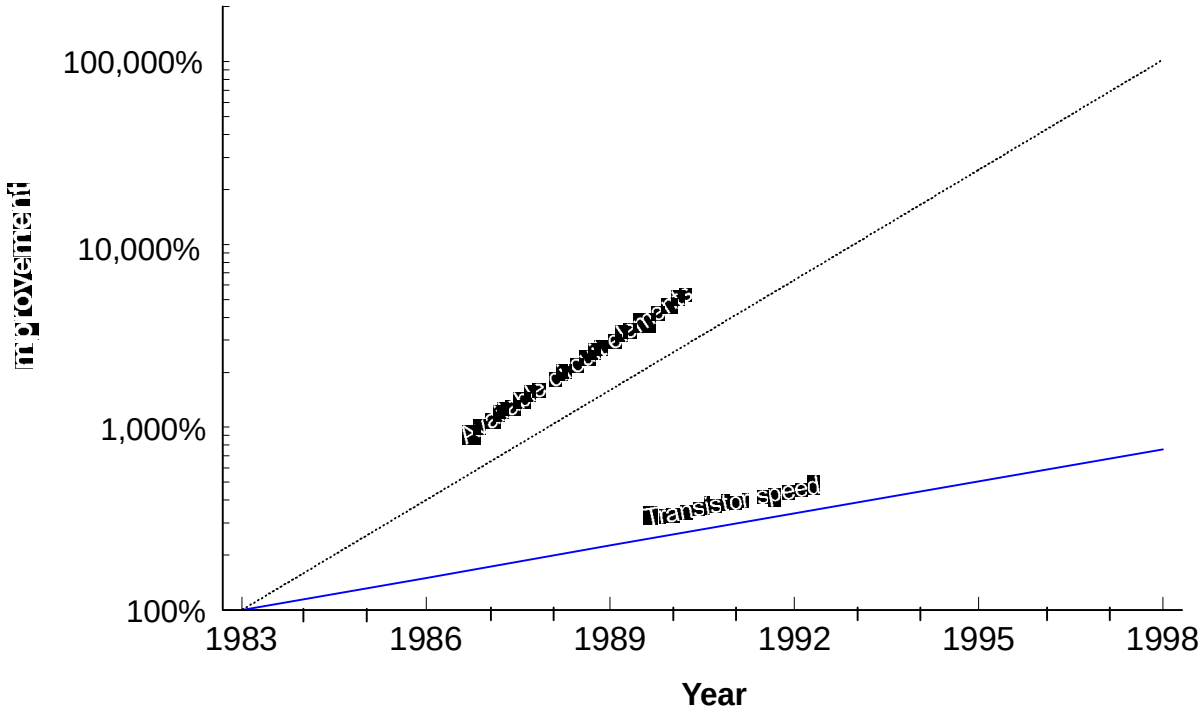

Figure 1: Using their 1983 capabilities as a baseline, transistor speed and the number of available CMOS circuit elements per die is plotted over time. Source: IBM and [1].

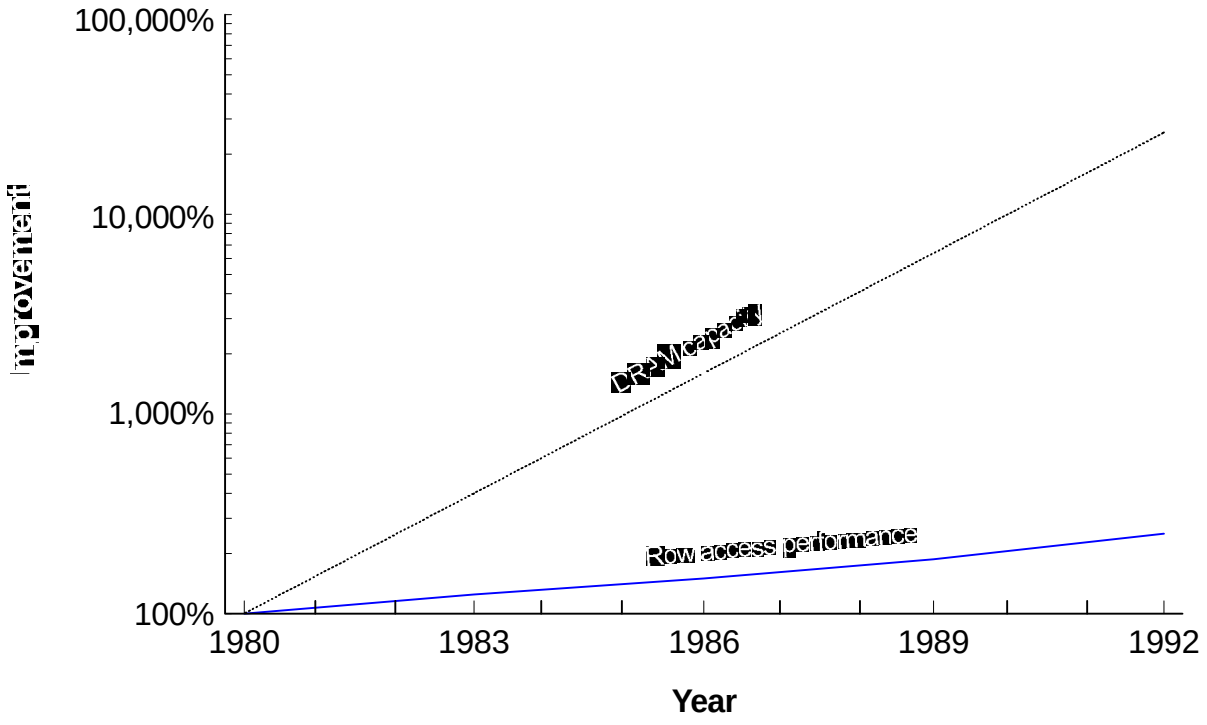

Figure 2: Using their 1980 capabilities as a baseline, the row access performance of DRAM and the DRAM capacity is plotted over time. Source: IBM and [2].

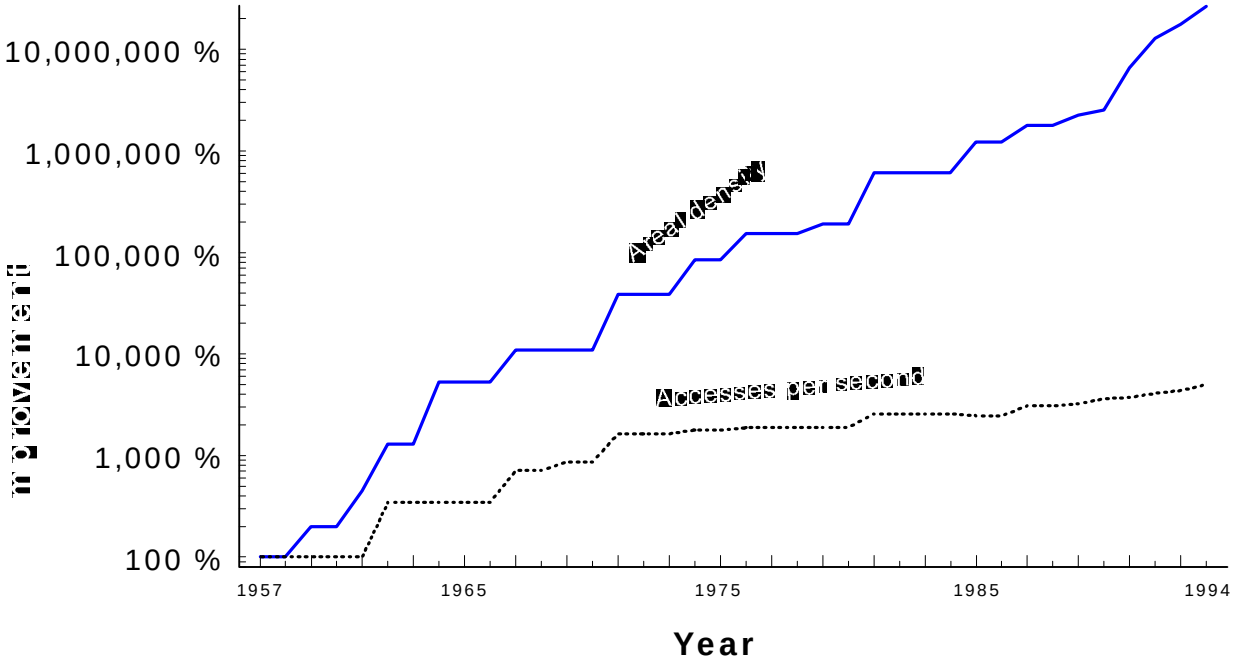

Figure 3: Using their 1957 capabilities as a baseline, magnetic disk areal density, and the average number of disk accesses per second are plotted over time. Source: IBM and [3].

In the earliest computer systems, both density and speed were quite modest; and software technologies were very immature. These facts drove computer architectures, which used very simple instructions and register sets combined with machine level programming, to provide the maximum in efficient exploitation of the limited number of switches and storage capacity. As hardware became denser and faster, the number of registers and the complexity of the instruction sets were increased. Advancing software technology allowed programming to move to the assembly language level and, eventually, to high-level procedural languages.

In the late 1950's and early 1960's, the gap was wide enough between memory technologies and processor speeds that functions implemented as instructions would execute as much as ten times faster than those programmed as function or subroutine calls. Compiler technology was still in its infancy and efficient compilation still eluded the general programming environment (though many early FORTRAN compilers were demonstrating quite astounding results for this time period). These technological constraints drove computer architects to provide ever more complex instruction sets. More complicated instructions helped overcome the memory-processor performance gap by doing more "work" for each instruction fetched. Many architects were also convinced that the job of the compiler could be simplified if more complex instruction sets could be used to close the "semantic gap" between the high-level languages and the machine-level instruction set.

As this trend toward more complex instructions continued into the 1970's, the underlying technologies were changing to the point where a fundamentally new set of design trade-offs was

possible for optimum system performance. Complex instructions were actually implemented using a high-speed, read-only control memory that actually interpreted the complex instructions and executed a "microprogram" contained in this high-speed storage. As instruction sets became more complex, this control store became larger and more complex as well. By the 1970's, the complexity reached a point where the correctness of this microcode could not be adequately assured. Thus, most computer systems began using a writeable control store so that corrections could be made in the field as errors were discovered and corrected. At about the same time, the concept of using high-speed, writeable memories, called caches, to contain the most recently used instructions and data proved quite effective in reducing the average latency (access time) to the memory subsystem because of the ability to exploit the temporal- and spatial-locality of most memory accesses.

Compiler technologies were advancing as well. Compiler optimization technologies were proving to be most effective when certain simplifying assumptions were used, including the use of very simple operations in the intermediate languages (IL) for performing the optimizations against. By using very simple IL operations, all operations were exposed to the optimization algorithms resulting in the dramatic reduction in the amount of code generated. These optimizations often resulted in code as good or better than that generated by average assembly language programmers. Another key set of technologies for compilers involved algorithms which allowed efficient utilization of the storage hierarchy. The use of registers has always been problematic for compilers. The register allocation problem was dramatically improved by the introduction of register coloring approaches [4]. The efficient exploitation of the cache and memory hierarchy became possible with the advent of cache-blocking techniques and loop transformations which enabled them.

From an operating system perspective, the trend towards portable operating systems, of which UNIX is the most obvious example, was creating a system design point in which most of the software in the system would be written in a portable, high-level language with relatively little code specifically written for the specific hardware system and almost no code written in assembly language. Combine all of these changes with the dramatically increasing density of the underlying hardware technologies and the fundamentals all pointed to a major change in computer system architectures. Several seminal research projects were undertaken in the mid to late 1970's based on these observed technological changes, including the IBM 801 [5], Berkeley RISC [6, 7], and Stanford MIPS [8] projects.

These projects challenged all of the existing fundamental concepts and approaches being used relative to the trade-offs between hardware and software. As often happens in major changes, these projects took the extreme opposite view of assuming that everything should be done in software unless proven otherwise. Thus, these projects explored all the boundaries of these design trade-offs. Fortunately, many of these proposed approaches never left the laboratory. For example, at one point the 801 project was attempting to eliminate the need for storage protection hardware by relying on a trusted operating system and compiler using a language which did not allow unbounded pointers. Theoretically this was a very nice concept, but not practical in the open systems software environment. Another example that all three projects explored was whether all instructions could be restricted to a single cycle operation. This implied that important functions like integer multiply and divide as well as almost all floating-point operations would have to be programmed from simpler,

one-cycle primitives. By the time these architectures were commercialized, these more "complex" operations were included. With the increased density of circuits available, the number of cycles required for these operations had reduced significantly (with many of today's RISC CPUs able to perform independent floating-point operations every cycle); and the overhead associated with making subroutine calls to special primitive instructions to perform these functions would have significantly impacted the overall performance for these operations.

The resulting commercialized RISC architectures included IBM POWER (based on the 801), Sun SPARC (based on Berkeley RISC), Silicon Graphics MIPS (based on Stanford MIPS), Hewlett Packard PA-RISC, and Digital Equipment Alpha. All of these architectures share some common characteristics. All have fixed-length (32-bit) instructions with relatively few formats making instruction decode extremely easy (sacrificing information density for ease of decode). Each has 32 general purpose registers as well as at least 16 floating-point registers. All have at least 32-bit addressing with extensions to 64-bit addressing (Alpha being the only one which supports only 64-bit addressing). Data references are restricted to explicit load and store instructions which transfer data from/to memory to/from an internal register. With the exceptions of operations such as integer multiply/divide and floating-point operations, the rest of their instruction sets are optimized around simple, single-cycle instructions. Separate instruction caches are used to supply the necessary instruction bandwidth and to provide a logical replacement for the high-speed microcode control store that previous machines used to contain microprograms. The advantage of an instruction cache is that the code for the most frequently used functions automatically migrates to the instruction cache as opposed to a microcode control store where the functions contained there are predetermined by the instruction set architecture.

These features all reflect the fundamental philosophy of creating a simple, straightforward instruction set that can be implemented efficiently in hardware and lends itself to efficient compiler optimization. Initial implementations of these architectures were able to achieve competitive instruction path lengths and to execute nearly one instruction every cycle for a wide variety of interesting workloads of which the SPEC benchmarks are representative. Workloads that are more stressful on the memory hierarchy, similar to that exhibited by the NAS parallel and TPC-C benchmarks, are dominated by cache and memory effects rather than the raw CPU capabilities and require careful attention to the design of the cache and memory subsystem. Nevertheless, the initial systems provided very attractive performance and price/performance, particularly in engineering/scientific applications, and established a significant foothold in the marketplace.

Some, less than optimal, trade-offs did survive the transition from the laboratory to commercialization. Most of these systems made some simplifying assumptions relative to cache management and coherency with I/O. The IBM POWER architecture took this to the extreme of requiring all cache-to-I/O coherency to be managed in software. While this has been made to work efficiently for the common case of paging I/O, the overhead for "raw" I/O is significant and complicated. Thus, cache-to-I/O coherency is now supported in hardware for PowerPC. The Sun SPARC architecture supports a feature called register windows which keeps the context of several layers of the subroutine-call hierarchy in registers. While this approach does reduce the overhead associated with subroutine calls and returns, it does restrict optimization across subroutines and

increases the overhead of context switching. The overall benefits are small and do not justify the additional hardware circuits and complexity.

After the initial round of implementations, computer architects realized that most future performance gains would have to come from the exploitation of parallelism. Since CPUs were already executing close to one instruction per cycle, without parallelism, almost all future performance gains would be limited to that obtained from faster circuit switching speeds. Exploitation of parallelism leverages the increasing circuit densities to achieve system performance growth which exceeds that of the underlying switching speed improvements. The forms of parallelism that have been explored include: instruction-level parallelism (executing more than one instruction per cycle), task/process-level parallelism (simultaneous execution of more than one task or process), and algorithmic parallelism (dividing up a single problem so that it can be executed simultaneously on more than one processor). Each of these approaches and their current status and future potential will be discussed.

Instruction-level Parallelism

After the initial RISC implementations, the focus of computer architects began to move from the current state of the art of nearly one instruction per cycle to multiple instructions per cycle (or superscalar implementations). This was driven by the fact that doing more operations in one cycle would allow the exploitation of additional circuit densities to increase system performance above that achievable by mere circuit switching speed improvements. Many of the architectural decisions made earlier mapped very well to superscalar implementations. The regularity of the instruction set and formats, the large number of general purpose registers, and the relatively few complex operations all greatly simplified the task of resolving interdependencies and resource conflicts necessary to dispatch and execute multiple instructions in one cycle.

However, some of the early RISC architecture decisions proved problematic in the move to superscalar implementations. The most obvious was the inclusion of the "delayed branch" in these architectures. Since a taken branch introduced a "bubble" of one cycle in the pipeline of these early processors, the delayed branch allowed the compiler to insert a useful instruction after the branch that could be executed during this otherwise idle cycle. Compilers were typically quite successful in finding instructions that could be placed in this delay slot; and this feature was very important for achieving the nearly one instruction per cycle goal of these systems. However, this feature did add complexity to the design since the delayed branch and its subsequent instruction often had to be treated as an atomic pair of instructions creating extra complexity especially under error conditions. On the other hand, superscalar implementations usually process branch instructions separately looking ahead into the instruction stream so that taken and untaken branches are essentially eliminated from the pipeline. Thus, not only is the delayed branch not needed, but it also adds complexity for no performance gain. This is why later RISC architectures like IBM POWER and DEC Alpha removed the delayed branch from their architectures.

Another area of concern with superscalar implementations is that performance tends to be ultimately limited by the ability to deal with the changes in control flow. Average instruction streams contain a taken branch every 5 instructions; so, to achieve any reasonable levels of parallel instruction execution requires significant amount of resources either resolving or predicting the outcome of branch instructions. Control flow is typically predicated on the testing of condition codes. Architectures which provide the ability for multiple, independent condition codes allow the compiler the opportunity to set and resolve them as early as possible in the code, greatly improving the accuracy of the branch resolution and prediction capabilities and simplifying what must be implemented for the same level of performance. Most RISC architectures were not designed with this in mind and have mixed capabilities in this area. Most have a single condition code, or worse, the MIPS and Alpha architectures combine the compare and branch into a single instruction. These architectures do provide the ability to place conditions into a general purpose register which is inefficient (only one bit of information in a 64-bit register), uses precious register locations that could be used for other computations, and adds complexity in branching logic which must then share these registers with the integer unit. The IBM POWER architecture is unique in having architected multiple, independent condition codes for use by the compiler as well as complete control of the setting of these condition codes in the instruction set.

Since the introduction of the first superscalar RISC, the IBM POWER microprocessors in the RISC System/6000 in 1990, increasingly aggressive superscalar implementations have appeared from all key vendors and have delivered even more impressive performance and price/performance particularly in engineering/scientific applications where the ability to execute fixed-point and floating-point instructions in parallel provides the ability to execute 3 to 6 instructions per cycle sustained with today's implementations. For a brief period, a few "superpipelined" processors were introduced in the market (most notably the MIPS R4000 and, to some degree, the original DEC Alpha which was both superscalar and superpipelined). The approach here was to exploit parallelism by making deeper pipelines instead of more parallel functional units. Since this approach does not exploit the extra density of circuits and puts extra stress on the circuit switching speeds, it has rapidly fallen out of favor.

Despite the success of superscalar RISC processors, the ability to efficiently exploit more instruction-level parallelism with purely superscalar approaches is reaching diminishing returns. This has given ascendancy to very long instruction word (VLIW) techniques [9]. The fundamental limit for superscalar designs is instruction dispatch. To exploit additional instruction-level parallelism requires an ever more complex instruction dispatch unit that can look past multiple branches and determine what instructions can be executed, or speculatively executed, on a given cycle based only on information available at run time. This is a geometrically complex problem as the number of instructions to be dispatched simultaneously increases. VLIW approaches move the determination of which instruction to execute, or speculatively execute, in a given cycle to the compiler based on compile-time information which is often more complete. The hardware implements a large number of parallel functional units, but only executes those operations in parallel that the compiler places into a single very-long-instruction word.

VLIW architectures which leverage large numbers of parallel functional units require more than the 32 registers typical of today's RISC architectures. They also fundamentally require the ability to resolve multiple branches in a single cycle implying the need for multiple condition codes.

Major advances in the state of the art of VLIW hardware and software have been made in recent years, but many issues remain unresolved. The biggest issues are compatibility with existing RISC architectures and the ability to support a range of VLIW implementations with the same compiled binary code. Now that RISC computers have established themselves in the industry, market acceptance of a new VLIW machine would be greatly enhanced by being able to support all existing applications without requiring a recompile. (In fact, this may be necessary for them to be commercially successful.) Also, a VLIW architecture must be able to support differing levels of parallelism in the underlying implementations to allow multiple price/performance points and to exploit increasing density in the future. This remains a field of very active research with all major players working to determine how to best exploit VLIW technology in their future systems.

Independent of VLIW or superscalar processors, the problem of "finite cache effects" is becoming an increasingly important aspect of system architecture and design. VLIW and superscalar processors can achieve very impressive levels of instruction-level parallelism as long as their data and instructions are contained within the high-speed caches connected to the processors. However, with the levels of parallelism being achieved, the number of cache requests per cycle is increasing; and the latency for resolving a cache miss is also increasing in relative terms. As figure 4 shows, the CPU performance is increasing at a faster rate than the access time of main memory creating an increasing performance gap. All of this is driving system designs which include multiple levels of caching; multiple, simultaneous outstanding cache misses; and cache line prefetching approaches. All of these focus on ways to decrease the average latency of cache misses and reflects incorporating parallelism and speculation into the cache subsystem to match the parallelism and speculation that is already in the processors. Algorithms and hardware implementations that are able to better tolerate latency will become increasingly important in the future.

Task/Process Parallelism

An obvious source of parallelism is the multiple, independent tasks and processes which run simultaneously on most computer systems. This level of parallelism has been exploited naturally by Symmetric Multiprocessor (SMP) systems by running independent tasks or processes on separate processors. This is a trend that will continue. Recently, interest has been revived in exploiting task/process parallelism at the individual processor level. Early CDC I/O processors were actually a single processor that was time-shared between multiple, simultaneous contexts on a

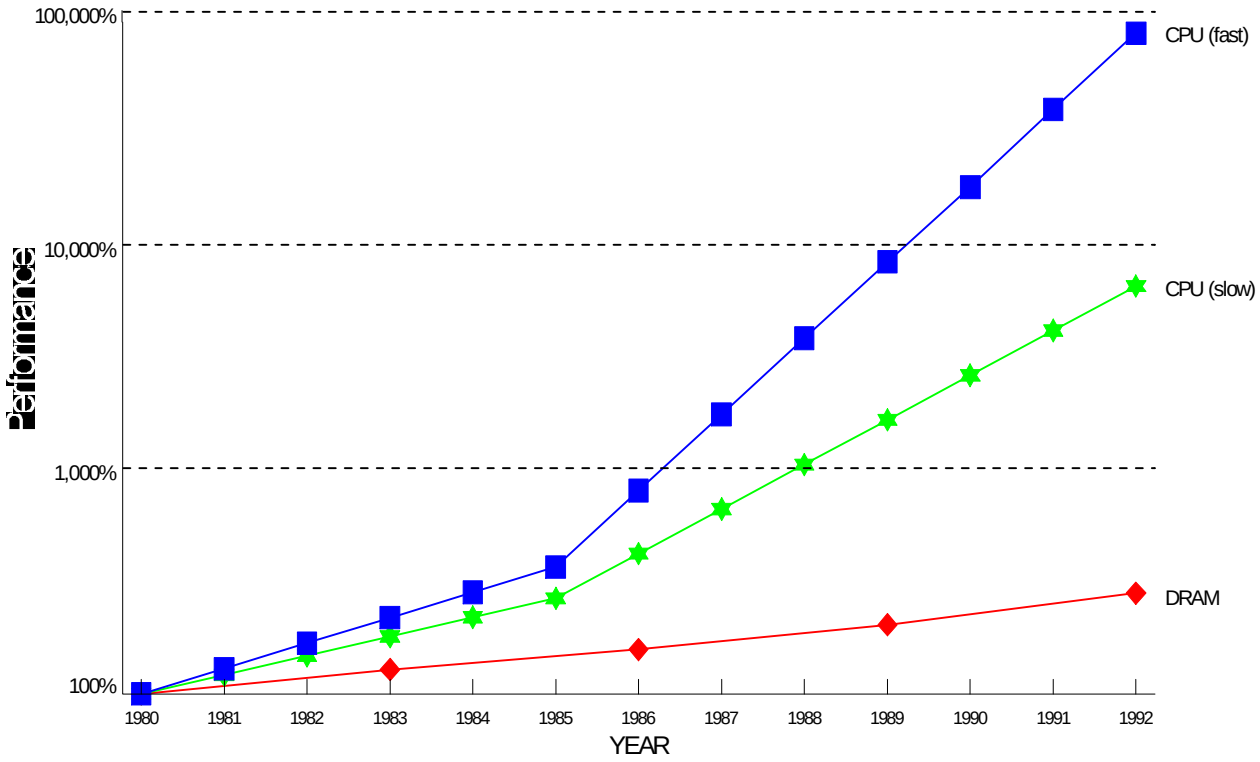

Figure 4: . Using their 1980 performance as a baseline, the performance of DRAMs and processors is plotted over time. The DRAM baseline is 64 KB in 1980, with three years to the next generation. The slow processor line assumes a 19% improvement per year until 1985 and a 50% improvement thereafter. The fast processor line assumes a 26% performance improvement between 1980 and 1985 and 100% per year thereafter. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. Source: [10].

cycle by cycle basis. In a similar fashion, much research is being done on sharing multiple contexts within a single processor with a very lightweight mechanism for switching contexts (on the order of a single cycle). Context switches would be initiated when a long running operation is encountered such as a cache miss. This would allow useful processing to be performed in other contexts during the latency of the cache miss, thus improving the efficiency of CPU utilization. For these approaches to be effective, the cache subsystems must be capable of supporting multiple, outstanding cache misses.

Algorithmic Parallelism

Task/process parallelism is able to improve the throughput of a computer system by allowing more tasks/processes to be completed per unit time. Algorithmic parallelism attempts to improve the turn-around time for a single problem or task, and may actually negatively impact throughput. Algorithmic parallelism involves rethinking the fundamental algorithms being used to solve a problem on a computer to come up with a new approach which will be able to be efficiently divided among several independent processors to work on the problem simultaneously. Some problems

require very little rework to accomplish this, though most require significant efforts to create an efficient and scalable approach. Almost all problems require significant redesign of algorithms to exploit high levels of multi-processor parallelism.

The fundamental limit of parallelism and scalability is the level of sharing required between the processors. Sharing can exhibit itself at the instruction, data, or I/O level. In actual implementations, sharing can also occur because buses, cache lines, or other implementation artifacts are shared between the processors, but are not sharing constraints imposed by the algorithms. Computer system designs are exploiting three approaches to reduce the amount of sharing at the hardware level, while still providing an efficient environment for the current state of the art in applications and operating systems. These three types of multiprocessors have been called: uniform memory access (UMA), non-uniform memory access (NUMA), and no remote memory access (NORMA).

Uniform Memory Access

This represents the sharing model of most commercially successful symmetric multiprocessors (SMPs). They are characterized by all processors having equal access to all memory and I/O. They typically run a single copy of the operating system that has been altered to exploit parallelism and to give the impression of a uniprocessor to the users and all uniprocessor applications. Applications that wish to exploit parallelism are provided with a set of programming interfaces, services, and tools to be able to execute on multiple processors simultaneously.

While UMA MP technology is quite mature, it does have some drawbacks for the exploitation of algorithmic parallelism. The fact that memory, I/O, and the operating system are all shared by all the processors creates a significant amount of implicit sharing which fundamentally limits the scalability of these designs. This problem only becomes worse as the speed of each individual processor increases. This will drive SMP systems to move from today's dominant design point of shared-bus systems to systems in which the connection among the CPUs and memory and I/O consists of various switched media to reduce the amount of implicit sharing. Further hardware enhancements will be added which speed up the cases where the algorithms themselves require sharing. Even with all these valiant efforts, the limit of scalability of UMA systems will remain in the low double digits.

In addition to performance scaling problems, UMA MP systems also suffer from system reliability problems. Because memory, I/O, and the operating system are all shared between all the processors, any failure in any component is highly likely to bring down the whole system. Thus, reducing the level of sharing can also improve the availability of a system, as well.

Non-Uniform Memory Access

NUMA machines share a common address space between all processors and their memory and I/O; however, the access time from any given processor to some of the memory and I/O in the system is noticeable longer. That memory and I/O with the shortest access time is usually referred to as "local" to that processor. Memory and I/O which is farther away is either referred to as "global" or "remote". Usually the bandwidth is higher to local resources than it is to remote resources.

By relaxing the requirement for uniform access, the hardware system designers are more easily able to construct systems which scale to much larger numbers of processors. Uniform access places significant physical constraints on the design since the speed of light ultimately determines access times. Thus, uniform access designs eventually reach a point where processors and memory cannot be packed more densely without increasing access times.

NUMA designs will function well provided most accesses can be made to local resources. In fact, the average percentage of remote accesses is the ultimate determinant of scalability in NUMA systems. Thus, the scalability of NUMA systems is highly dependent on the ability of the application and the operating system to reduce sharing. Some of this is accomplished by replication. For read-only data (or instructions) that are shared, a replicated copy can be placed locally everywhere it is being used and all references can be made local. Data (or instructions) that are not shared only exist locally by default. The challenge comes in reducing the amount of sharing of data that is updated and read by more than one processor.

Further complicating the situation is the fact that since all addressing is common, but data is allocated in discrete sizes (such as pages or cache lines), false sharing can occur by having data that is not being shared having been allocated to addresses too close together. Also, the realities of trying to balance processing loads among the multiple processors requires that processes or tasks must sometimes be migrated to other processors. This migration requires that the associated local data be moved also. Since the operating system and hardware are attempting to manage this at run time with only run-time information, this is a difficult task in the general case.

The design of the operating system for NUMA systems has two approaches. The first is to modify an operating system designed for UMA machines to deal with replication of memory and remote/local optimizations. This will ultimately entail replicating many operating system services to more distributed approaches such as having local task dispatch queues coordinated by an overall global task dispatch algorithm. The advantage of this approach is that a single system image is preserved that helps applications in the transition from UMA machines. This is the approach that has been taken by most NUMA machines in the marketplace.

A second approach is to run separate copies of the operating system locally (i.e. replication of the operating system); and then provide global services that each operating system copy uses to manage the resources of the whole system. This approach has the advantage of starting from a design point where no sharing exists and adding sharing and cooperation as needed through the global services. The downside is that since all services are not global, the migration from UMA machines is

complicated. Furthermore, modifying the operating system to be able to cooperate on controlling global resources is usually non-trivial.

The common thread here is that NUMA machines provide scalability by reducing sharing through the use of replication. Since replication explicitly uses more memory, this is another example of exploiting density improvements to improve performance.

No Remote Memory Access

NORMA machines do not share memory, I/O, or operating system copies. The individual nodes in a NORMA machine cooperate with other nodes through the sending of messages. High-performance NORMA machines focus on improving the bandwidth and latencies of sending messages between nodes particularly at the application level. Much like the second approach mentioned for NUMA operating system design, global services and resource allocation are provided by a layer of software which coordinates the activities of the multiple copies of the operating system. Applications must be structured and written to exploit message passing as opposed to shared-memory communication constructs and to use only those services that are globally enabled by the coordinating layer.

The NORMA approach is clearly the most scalable from a hardware perspective and has demonstrated this capability for applications which have been structured to have limited sharing, as well. However, most applications have yet to make this transition. To assist in this complex process and to improve performance, NORMA machines are beginning to provide "remote memory copy" operations which allows an application to copy to/from another application's address space in another node on the system in a very lightweight fashion. This capability will further close the semantic gap between shared-memory programming and message-passing programming and allow compilers and application writers to be able to more easily design and move applications to NORMA machines. Also, the move to object-oriented application design lends itself well to NORMA architectures since method invocations are essentially messages.

In fact, if one looks at what is required to implement an efficient NUMA machine and a NORMA machine that supports remote memory copy, much of the hardware required is quite similar. Both machines consist of nodes with local memory and I/O interconnected through some kind of fabric to other nodes. In a NUMA machine, data is transferred from one node to another based on a cache miss to a remote node. In a NORMA machine, data is transferred between nodes at the request of an application (with a PUT or GET command). The actual process of doing the data transfer is very similar in hardware. The key difference is whether the transfer was initiated by hardware (a cache miss in a NUMA machine) or software (a PUT/GET in a NORMA machine). Because of these similarities, some of the current research projects are attempting to design machines that are capable of supporting both simultaneously.

Again, NORMA machines achieve performance by exploiting replication, even more so than NUMA machines. This, of course, leverages increasing density for more performance.

Summary

Computer architectures have evolved to optimally exploit the underlying hardware and software technologies to achieve increasing levels of performance. Computer performance has increased faster than the underlying increase in performance of the digital switches from which they are designed by exploiting the increase in density of digital switches and storage. This is accomplished by the use of replication and speculative execution within systems through the exploitation of parallelism. Three levels of parallelism have been exploited: instruction-level parallelism, task/process-level parallelism, and algorithmic parallelism. These three levels of parallelism are not mutually exclusive and will likely all be used in concert to improve performance in future systems.

The limit of replication and speculation is to compute all possible outcomes simultaneously and to select the right final answer while discarding all of the other computations. This is analogous to the approach DNA uses to improve life on our planet. All living entities are a parallel computation of which only the "best" answers survive to the next round of computation. These are the concepts driving research in the area of using "biological" or “genetic” algorithms for application design as well as the research into using DNA to do computations. In the limit, molecular-level computing provides huge increases in density over current systems, but much more limited improvements in raw performance. Thus, we can expect continued advancements in those approaches and architectures that exploit density and parallelism over raw performance.

References

[1] R.F. Sechler, and G.F. Grohoski, ‘Design at the System Level with VLSI CMOS’, *IBM Journal of Research and Development*, 39.1-2, pp. 5-22 (1995).

[2] D.A. Patterson, and J.L. Hennessy, *Computer Architecture, A Quantitive Approach*, Morgan Kaufmann Publishers, San Mateo, CA (1990).

[3] J.M. Harker, D.W. Brede, R.E. Pattison, G.R. Santana, and L.G. Taft, ‘A Quarter Century of Disk File Innovation’, *IBM Journal of Research and Development*, 25.5, pp. 677-689 (September 1981).

[4] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein, 'Register Allocation via Coloring', *Computer Languages*, 6.1, pp. 47-57 (1981).

[5] G. Radin, 'The 801 Minicomputer', *SIGARCH Computer Architecture News*, 10.2, ACM, pp. 39-47 (March 1982), Revised version published in *IBM Journal of Research and Development*, 27.3, pp. 237-246 (May 1983).

[6] D.A. Patterson, and D.R. Ditzel, 'The Case for the Reduced Instruction Set Computer', *SIGARCH Computer Architecture News*, 8.6, ACM, pp. 25-33 (October 15, 1980).

[7] D.A. Patterson, 'Reduced Instruction Set Computers', *Communications of the ACM*, 28.1, pp. 8-21 (January 1985).

[8] J.L. Hennessy, 'VLSI Processor Architecture', *IEEE Transactions on Computers*, C-33.12, pp. 1221-1246 (December 1984).

[9] J.A. Fisher, 'Very Long Instruction Word Architectures and the ELI-512', *The 10th Annual International Symposium on Computer Architectures*, ACM, pp. 140-150 (June 13-17, 1983).

[10] J.L. Hennessy, and D.A. Patterson, *Computer Organization and Design, The Hardware/Software Interface*, Morgan Kaufmann Publishers, San Mateo, CA (1994).