A Look at Software Performance Since 1940s

Igor Zinkovsky (igorzi@microsoft.com)
Nikolay Topilski (ntopilsk@cs.ucsd.edu)

Abstract

A look though the history of computing shows very rapid growth of various technologies enabling computer industry to become the dominant force it is today. Computers went from being immensely bulky, unreliable, and expensive to elegant, connected, and affordable. Throughout this evolution process one important aspect always occupied the minds of system and application engineers - performance. This paper will go through the history of computing beginning with 1940s and highlight important aspects which defined how computer performance was approached at each significant milestone of computing.

Early computers were custom-made electronic digital machines commissioned by the military to perform intense numerical calculations. Important aspect of these early machines was the ability for people to program, making its users to be first computer programmers. Although the concept of software has not been born yet and programs consisted of switches-wires combinations, we already see programmers think about optimizing programs to work around performance bottlenecks.

During 1950s-60s computer industry went through one of the most significant transitions in its short history. Due to significant advances in hardware, computers were becoming increasingly powerful, and as a result computer applications were becoming increasingly complex. To deal with complexity, we see a sharp separation of hardware and software. Ever since the separation the goal of software performance engineering has been to develop creative solutions to reduce or tolerate speed limitations placed on the system by hardware. This principle is being followed by the industry until this day. For example, disk and memory latencies have always been most noticeable performance bottlenecks. While the reduction and tolerance of latency involves hardware components, we also see software in constant evolution to work around these issues.

Another important aspect is the tradeoff the software industry is willing to make for increased developer productivity, which usually comes at a system or application performance penalty. This is highlighted by the development of compilers, libraries, abstraction layers, and execution frameworks. All of these innovations have helped to significantly improve developer productivity, which in turn helps software vendors ship products faster. In almost all cases, the software industry comes to a consensus that taking a reasonable performance hit for making developers more productive is perfectly acceptable. User applications for PCs have enjoyed this trend through early 2000s, while semiconductor manufacturers were able to double the CPU clock speed every 18 months. Since 2004 the semiconductor industry has not delivered a CPU with significant improvement in CPU clock speed. Moore's law is expected to be followed for at least another decade, and CPU makers have decided to begin producing chips with multiple

CPU cores. Unlike CPU clock growth multiple cores will no longer allow software to automatically run faster with release of new wave of CPUs. Instead software industry must find a way to adopt concurrency in its design of user applications.

Early Machines (1940s)

1940s witnessed the development of first fully electronic digital computers. All of such machines were very large scale expensive creations. They were primarily commissioned by the military, which had the need for computing power and was able to afford it. Such computers were designed with primary goal of performing high volume of similar calculations. In our discussion we will concentrate on one such machine, ENIAC, and examine the problems and trade-offs associated with its design and programming.

ENIAC, which is acronym for Electronic Numerical Integrator and Computer, was first large-scale, fully electronic, digital computer with reprogramming capability, which enabled it to solve a range of problems. ENIAC was commissioned by U.S. Army Ballistics Research Laboratory to calculate artillery firing tables, however first calculations that were run on the machine were related to the design of the hydrogen bomb. It was unveiled in February of 1946 with the price tag of $500,000.

The ENIAC proper consisted of 40 panels arranged in U shape. It had three portable function tables, a card reader, and a card punch. The units of the ENIAC were functionally classified into four categories: arithmetic, memory, input and output, and governing. The unit of the ENIAC referred to one or more panels and associated devices (such as, the portable function tables) containing the equipment for carrying out certain specific related operations. The arithmetic units included 20 accumulators (for addition and subtraction), one high-speed multiplier, and one combination divider and square rooter. The constant transmitter, three function tables, and the 20 accumulators provided numerical memory. The constant transmitter with associated card reader read from punched cards numbers that were changed in the course of a computation and made these numbers available to the computer as needed. The constant values were stored on the switches of the constant transmitter or of the portable function tables and were supplied when needed.

ENIAC was capable of performing arithmetic operations of addition, subtraction, multiplication, division, and square rooting on signed numbers expressed in decimal form. ENIAC used ten-position ring counters to store digits. Arithmetic was performed by "counting" pulses with the ring counters and generating carry pulses if the counter "wrapped around", the idea was to emulate in electronics the operation of the digit wheels of a mechanical adding machine. Once configured to follow a routine consisting of operations in its repertoire, ENIAC carried out the routine without further human intervention. The machine could be instructed to carry out complex mathematical operations of interpolation and numerical integration and differentiation. The results were recorded on punched cards.

The “clock” frequency of ENIAC was 10 microseconds. The fastest operation was addition, which executed in 20 cycles, allowing for 5000 additions to be executed per second. All other operations required an integral number of addition times, and as such, were measured in units of add-time. Despite slow operations, ENIAC could be configured to run the calculation in parallel, achieving much faster peak speeds.

Nevertheless, before any computation could be performed on ENIAC, the machine required an extensive physical configuration. Input and output terminals of the units had to be connected into busses for communication of numerical data. Also, each of the units had to be setup as to recognize when they are to operate and which particular operations had to be performed. Program controls and program trays and cables were used to configure ENIAC for a particular computation. Each unit of the ENIAC had one or more program controls. Each program control for a unit that was capable of more than one operation or which was capable of performing operation in a variety of ways had a set of program switches of its own. As result, the machine setup, because of its complexity, was a science in itself and required extensive periods of time to accomplish. Usually a setup took one or more days.

Another issue of concern was testing proper functionality of ENIAC. The machine was very complex. It had about 20,000 tubes and thousands of switches and plug-in contacts. Since any of these or other things could fail, it was not surprising that the duration of an average run was only a few hours. A power failure could spoil several tubes. Some of the failures were not clear-cut, that rendered a vacuum tube inoperable, but rather caused it to operate improperly once in a hundred to a thousand times. As result, half of the operation time was spent on testing.

Complexity of the programming and testing of ENIAC prompted several improvements to be made from 1948. A primitive read-only stored programming mechanism using the Function Tables as program Read Only Memory (ROM) was introduced. Three digits of one accumulator were used as the program counter, another accumulator was used as the main accumulator, and another accumulator was used as the address pointer for reading data from the function tables. Also, a converter and register devices were added to alleviate negative impact on speed of the above changes. The converter converted any two-digit number to a program pulse in one add-time, thus freeing up cycles for other operations. The register device was a set of one hundred ten-digit registers with three operations: It could clear its old argument and receive a new argument in one add-time; it could receive a number, clear that register designated by its current argument and send the number to that register, increasing its argument by one; it could send out (and also keep) the number designated by its current argument.

The implemented hardware changes achieved several drastic improvements in programming and functional testing:

1 The programming ENIAC became clearer and straightforward. It allowed scientific personnel with no prior experience with computing machines to code their own problems and prepare tests.

2 Not only programming of ENIAC became easier, it became faster and allowed for problems four times larger to be programmed on the machine.

3 The problems were input in the new system by setting switches on the function table, instead of the old method of plugging numerous cables. The computation setup was accomplished in hours instead of days. Furthermore, the switch settings made in the function tables could be systematically and rapidly debugged.

Besides easing the switch settings check, the new method eased the ENIAC functional testing. Testing was of outmost importance, and it was done thoroughly, rapidly and systematically without touching any cables or switches. On one function table, programs were set up at will to test any suspected unit. Thus, the implemented improvements in programming and testing attributed to the ENIAC delivering results at a greater rate despite the reduction in computing speed by a factor of six.

It was also observed that ENIAC remained I/O bound because of the difference of electronic speed of computation and electromechanical speed of input and output generation. I/O remained to be the bottleneck for practical real world problems even after the speed reduction from the modifications described above.

Early machines II: (1950s – 1960s)

During 1950s computer industry went through important improvements in the area of hardware that allowed for significant leaps in software development. For the purposes of the discussion we will analyze a sample machine from the period, made by IBM which was a dominant computer hardware manufacturer of the time.

In early 1950’s IBM introduced machines of the 700 series. The 700 series were large-scale electronic digital computers. IBM 701 was formally announced in May of 1952 and was IBM’s first commercial scientific computer. Its business counterpart was IBM 702. The computers and associated components were not bought but rather rented from the manufacturer. The price for the rental of the main analytical unit was $8000 per month, with other components costing slightly less. Even though the cost of using computers dropped comparing to previous decade, the computing time remained very expensive. As result high utilization of the machine remained a priority.

The 701 contained the arithmetic components, the input and output control circuitry, and the stored program control circuitry. Also, mounted on the 701 was the operator's panel. The arithmetic section contained the memory register, accumulator register and the multiplier-quotient register. Each register had a capacity of 36 bits. The 36 bits was the necessary number of bits to represent signed integers to an accuracy of ten decimal digits. It also allowed for storage of six alphanumeric characters encoded in a six-bit character encoding. The accumulator register also had two extra positions called register overflow positions. The system used electrostatic storage, consisting of 72 cathode ray tubes, three inches in diameter each. The tubes provided capacity of 1024 bits each, giving a total memory of 2048 word of 36 bits each. Memory could be expanded to a maximum of 4096 words of 36 bits by the addition of a second set of 72 tubes. Another option was to replace two of the electrostatic storage units (IBM 706) with magnetic core storage unit (IBM 737) of equal capacity. The magnetic core offered faster access times of 12 microseconds versus 30 microseconds of electrostatic storage. The functional machine cycle of the 701 was 12 microseconds; the time required to execute an instruction or a sequence of instructions was an integral multiple of this cycle. A whole word addition operation took 60 microseconds, 5 cycles, 456 microseconds or 38 cycles were required for execution of multiply or divide instructions. All operations of the Type 701 were controlled by a program stored in a single address space.

The usual method of input to the system was magnetic tape, but entry could also be gained from punched cards through the card reader or from the operator's console, if special instructions were required. All information, whether part of the data to be processed or part of the program of instructions, started out on punched cards. Then, it either could be converted directly to magnetic tape before being read into the system or it could be read directly.

The control section decoded the stored program and directed the machine in automatically performing its instructions. Instructions were entered into the control section through storage or manually from the operator’s panel. The entire machine could be controlled from the operator’s panel through various buttons, keys and switches. The operator could manually control the insertion of information into various registers. The content of various registers could also be displayed in neon lights for examination. The operator’s panel was primarily used for the initial testing of a program for new operation and the beginning of the program execution.

The results of the processing were either produced on a line printer, on a magnetic tape, or on punched cards. If the operator did not want to tie up the entire system while the relatively slow printing or punching was accomplished, he or she could produce an output tape, then connect a tape unit directly to the printer or card punch and print out or punch the results without using the principal components of the system.

What is of interest is the programmability of the machines. The earlier computer systems were programmed by hand, via physical configuration of cables and front panel switches that were used to enter instructions and data. The switches represented the address, data and control lines of the computer system. To enter data into memory, the address switches were toggled to the correct address, the data switches were toggled next, and finally the write switch was toggled. This wrote the binary value on the front panel data switches to the specified address. Once all the data and instruction were entered, the run switch was toggled to run the program. The programmer also needed to know the instruction set of the processor. Each instruction needed to be manually converted into bit patterns, so that the front panel switches could be set correctly. Obviously, this methods were slow and error prone.

The improvements in the production techniques of memory modules allowed for increases in memory sizes and lower costs. The larger memory sizes allowed for a logical next step of writing a program to interpret another. The new interpreter, assembler, would be run by the computer, and translate the actual mnemonics of the assembly language into operation codes of machine instructions, the task previously done manually by programmers. Furthermore, it lowered the entry barrier to become a programmer, removing the need to know instruction codes. The new approach was vastly superior to previous method. It eliminated translation errors, greatly sped up the conversion to machine code and allowed for easier error checking and changes. The net effect was faster program development cycle. Thus, the development for IBM 701 could be done in assembly.

In April of 1954 IBM introduced IBM 704, the first mass produced computer with floating point arithmetic hardware. The Type 704 was twice as fast as its predecessor IBM 701. IBM 704 multiplies or divides took 240 microseconds, or approximately 4,000 operations per second. The usage of cathode ray tubes was completely abandoned in favor of magnetic core memory. Three index registers were added to allow for indirect memory addressing. To support these new features the instructions were expanded from 18 bits to full word length of 36 bits.

In addition to this high-speed memory, the 704 had a magnetic drum reader and recorder. Each of the two drums of the magnetic core could hold 4096 words, for the total capacity of 8192 words of permanent storage with 50 milliseconds of random access time. The drum could be used for storage of parts of the program, intermediate results, rate tables, or other information. Finally, the magnetic tape unit acted as bulk storage as well, each holding up to 5 million characters.

Even though automation of assembler greatly simplified programming there remained several important inherent disadvantages, which remain true to this day:

1 The programmer requires knowledge of the processor architecture and instruction set.

2 Source programs tend to be large and difficult to follow, since many instructions are required to achieve small tasks.

3 Programs are machine dependent, requiring complete rewrites if the hardware is changed.

The idea of high level programming languages existed in early 50's, for example the seminal work by Grace Hopper on the A-0 programming language, but the machines of the early 50s were still not powerful enough to allow for their realization. FORTRAN team led by John Backus at IBM is generally credited for having introduced the first complete compiler, in 1957 on IBM 704.

A compiler is itself a computer program written in some implementation language. Early compilers were written in assembly language. Also, because of the existing memory limitations the compilers of the time were split into smaller programs each of which made a pass over the source or some representation of it, performing required analysis and translations.

A draft specification for *The IBM Mathematical Formula Translating System* was completed by mid-1954. The first manual for FORTRAN appeared in October 1956, with the first FORTRAN compiler in April 1957. FORTRAN offered a considerable reduction in the training required to program, as well as in the time consumed in writing programs and eliminating errors. It had an optimizing compiler and produced object programs which were nearly as efficient as those written in assembly language by human programmer. Otherwise customers would have been reluctant to use it. One of the features introduced by FORTRAN was ability to perform a repetitive task from a single set of instructions by using loops.

The language was widely adopted by scientists for writing numerically intensive programs, which further encouraged compiler writers to produce compilers that could generate faster and more efficient code. The inclusion of a complex number data type in the language made FORTRAN especially suited to technical applications such as electrical engineering. In 1958 FORTRAN II was released. It was a significant improvement. It added the capability for separate compilation of program modules and dynamic linking of assembly modules. FORTRAN IV was released in 1961. It offered 'cleaned up' version of FORTRAN II code and eliminated some machine-dependant language irregularities. Another major step forward was taken on May of 1962, when an American Standards Association (ASA) committee started developing a standard for the FORTRAN language, a very important step that made it worthwhile for vendors to produce FORTRAN systems for every new computer, and made FORTRAN an even more popular high level language. The new ASA standard was published in 1966, and was known accordingly as FORTRAN 66, it was the first high level language standard in the world.

Parallel to FORTRAN there were other high level languages being developed. Particularly COBOL, Common Business Oriented Language, was designed for business use. Early COBOL efforts aimed for easy readability of computer programs and as much machine independence as possible.

The idea of compilation quickly caught on, and most of the principles of compiler design were developed during the 1960s. Thus, while I/O still remained the bottleneck, the expansion of memory capacity allowed for advancements in software development granting faster development of larger and more complicated portable programs. Greatly reducing development and debugging times brought down the overall costs of computer utilization and increased the scope of addressable problems.

Mainframes and Minicomputers

Many innovations of 1960s have made it possible to drastically increase business applications for computers. Because computer ownership (or leasing) costs were very high, businesses were willing to invest significantly in performance tuning to maximize their investment. From business perspective it was very important that the computer's processing power was fully utilized.

There were many advances in I/O devices in 1960s. Disk and tape drives were developed, and card readers/punchers continued to be widely used. Most business applications manipulated large amounts of data, and were largely I/O bound. This is in contrast to supercomputers which were primarily used for scientific compute-bound problems. With moving mechanical parts disk drives inherently possess high latency for random I/O requests. This is still a major bottleneck in system performance even today, and it is a job of many software architects and developers to engineer applications in a way to reduce random I/O. Even when not counting latency as a factor, disk/tape transfer rates are far slower compared to CPU.

Because I/O devices were recognized as sources of system bottlenecks, it was necessary to design a mechanism where the CPU would still be utilized while a program waited for a response from I/O device. This was achieved using multiprogramming, where several programs were loaded at the same time, and the executing program would give up the CPU when it needed to wait for an I/O operation. IBM introduced this concept with its MFT (multiple tasks of fixed size) operating system, which was later followed by MVT (multiple tasks of variable size). The operating system stored away the context of a program when it reached an instruction requiring I/O, and loaded the context of another program ready to be executed.

Many programs requested input from the user, which usually provides slower response time than an I/O device. Letting only one user interact with a machine where the CPU is almost always idle would be very expensive and impractical for most businesses. Time sharing systems solved this by letting several users connect into a mainframe machine using terminals. This allowed operating systems to treat users as I/O devices, and use multiprogramming techniques to maximize the CPU utilization.

In late 1970s, two seconds was the acceptable response time for time sharing systems. It was largely believed that two seconds were acceptable because the person used it to think about next tasks that he/she would need to enter. However, the "The Economic Value of Rapid Response Time" paper (by Walter J. Doherty, IBM and Ahrvind J. Thadani, IBM) describes a study done to show that a person typically has a sequence of actions in mind, and reducing the response time will result in more transactions between the person and the computer per unit of time. This translates into a more productive working environment, and cuts business costs. The study shows that the number of transactions a user completed in an hour goes up dramatically when response time is less than two seconds. For example, with 3 seconds response time the study found that the user executes about 180 transactions per hour; however with 0.3 seconds response time the number of transactions went up to 361 (106% increase). In other words, 2.7 seconds of response time saved 10.3 seconds of user time. What's even more interesting is study shows that an average experienced engineer working with subsecond response time was as productive as an expert engineer working with slower response time. Finally, the study shows that users have higher enthusiasm working on systems with quicker response time, which directly translates into higher quality of work.

Advances in multiprogramming and time sharing resulted in more programs loaded in memory at the same time. The amount of physical memory was very limited, and fitting all needed programs in it was not practical. The development of virtual memory allowed developers to write programs against a virtual address space instead of physical memory, which supported multiprogramming and allowed far greater flexibility. Virtual memory didn't come without performance cost. Virtual to physical address translation worried many in the industry due to being potentially expensive. Most operating systems use paging as part of virtual memory implementation. Paging has opened a new class of performance issues. When a requested page is not in physical memory, a page fault is issued, and its resolution requires issuing an I/O request. In many cases applications are bound by the number of page faults they experience. Minimizing the number of page faults in applications is still a goal for many software architects today.

As previously stated software must deal with hardware's limitations and adapt to running in a constrained environment to maximize resource utilization. While hardware support for advances such as multiprogramming and virtual memory is required, software's flexibility is what really allowed these primitives to take computing power to the next level.

As we have already seen, operating systems started supporting MFT and MVT to allow maximum CPU utilization. In this model the operating system must decide which program will be given the CPU when currently executing program makes an I/O request. Such decisions make up scheduling policies, which can be implemented exclusively in software, and easily plugged-in depending on the types of applications the system is expected to execute.

With virtual memory and paging, the operating system must also make similar decisions about what memory pages should be swapped out to make room in physical memory for newly requested pages. Such decisions are classified into policies, and are implemented in software, which again allows them to be plugged-in for specific purposes.

At the application level software developers focused on minimizing the number of I/O operations as well as making those operations sequential rather than random. For time sharing systems, the response time was the metric used to assess performance, and as we have already seen, there was substantial business case for making response time quicker.

The computer industry was making significant investments during 1960s-70s to maximize the utilization of very expensive machines. We see these investments being spread out all across the industry. System manufacturers innovated with new approaches at hardware and operating system levels to allow applications to fully utilize the CPU. Businesses that relied heavily on mainframe computers made significant investments at increasing throughput and response time (on time sharing systems) of their business applications. During this time, many engineers started to specialize in tuning systems and applications to maximize performance. Consulting companies started forming to help businesses optimize their applications.

Early Personal Computers (1970s – 1980s)

In order for personal computers to be developed a significant advancement had to take place in hardware. Such advancement was the development of the microprocessor. As with many advances in technology, the microprocessor was an idea whose time had come. Prior to microprocessor invention, electronic CPUs were typically made from bulky discrete switching devices or small-scale integrated circuits, containing the equivalent of only a few transistors. By integrating the processor onto one or a very few large-scale integrated circuit packages, containing the equivalent of thousands and later millions of discrete transistors, the cost of processor power was greatly reduced. Since the advent of the integrated circuits, the microprocessor would become the most prevalent implementation of the CPU, nearly completely replacing all other forms.

What is widely accepted to be the first commercial single chip microprocessor was Intel 4004. Intel 4004 had 4-bit processor meant for a calculator. It processed data in 4 bits, but its instructions were 8 bits long. Program and data memory were separate, 1K data memory and a 12-bit PC for 4K program memory. There were also sixteen 4-bit or eight 8-bit general purpose registers. The 4004 had 46 instructions, using only 2,300 transistors. It ran at a clock rate of 740 kHz, eight clock cycles per CPU, with the cycle of 10.8 microseconds. In 1972 Intel released Intel 4040, which was an enhanced version of the 4004, adding 14 instructions, larger stack, 8K program space, and interrupt abilities, including shadows of the first 8 registers.

Intel offered Busicom, the calculator manufacturer, a lower price for the chips in return for securing the rights to the microprocessor design and the rights to market it for non-calculator applications. Busicom, in financial trouble, agreed.

Texas Instruments followed the Intel 4004/4040 closely with the 4-bit TMS 1000, which was the first microprocessor to include enough RAM, and space for a program ROM, and I/O support on a single chip to allow it to operate without multiple external support chips, making it the first microcontroller. It also featured an innovative feature to add custom instructions to the CPU. It included a 4-bit accumulator, 4-bit Y register and 2 or 3-bit X register, which combined to create a 6 or 7 bit index register for on chip RAM.

The 8080 was the successor to the 8008, similar to 4040 design, 8008 was released in April 1972 and was intended as a terminal controller. While the 8008 had 14 bit PC and addressing, the 8080 had a 16 bit address bus and an 8 bit data bus. Internally it had seven 8 bit registers, a 16 bit stack pointer to memory which replaced the 8 level internal stack of the 8008, and a 16 bit program counter. It also had 256 I/O ports. As such, I/O devices could be hooked up without taking away or interfering with the addressing space.

Shortly after Intel's 8080, Motorola introduced the 6800 microprocessor. Some of the designers left to start MOS Technologies, which introduced the 650x series which included the 6501 and the 6502. Like the 6800 series, variants were produced which added features like I/O ports or reduced costs with smaller address buses. The 650x was little endian and had a completely different instruction set from the big endian 6800.
Unlike the 8080 and its kind, the 6502 and 6800 had very few registers. It was an 8 bit processor, with 16 bit address bus. Inside, there was one 8 bit data register, two 8 bit index registers, and an 8 bit stack pointer. At the time, RAM was actually faster than microprocessors, so it made sense to optimize for RAM access rather than increase the number of registers on a chip. It also had a lower gate count and cost than its competitors.

Microprocessors made possible the advent of the microcomputer in the mid-1970s. It also allowed for the new form of computers to be born. The home or otherwise known as personal computers were made possible by significant reduction in processor manufacturing costs, which translated to lower prices. What is accepted to be the first personal computer was MITS Altair 8800, a microcomputer design based on Intel 8080 CPU. Altair 8800 was announced in January of 1975 and sold as a kit that had to be assembled by users. Altair’s user interface consisted of the front panel filled with lights and switches. The switches allowed direct input of the machine code into registers. There was no way to store programs, and as such they had to be entered anew after every power down. Despite of the limited repertoire Altair 8800 gained popularity among computer hobbyists and several thousand units were sold.

Despite of the relative success of the Altair, the established electronics and computer manufacturers did not consider personal computers as a viable market that could generate profit. It took up and coming company of Apple Computers Inc. to produce Apple II, the machine that started personal computer revolution.

The first Apple II computers were demoed at the West Coast Computer Fair in 1977 and went on sale on June 5, 1977. Apple II looked like an appliance rather than a piece of electronic equipment, having been placed into a plastic case with the keyboard. Apple II utilized MOS Technology 6502 microprocessor, primarily because of its low cost, running at 1 MHz and 4 KB of RAM. It also boasted high-resolution graphics modes, sound capabilities, Integer BASIC programming language interpreter, also written by Wozniak, and placed into the ROMs and audio cassette interface for loading programs and storing data. The programs that could be written in BASIC were loaded directly into RAM and executed. Due to absence of virtual memory, programs needed to be constrained in size in order to fit into the available physical memory. Therefore optimizing programs for size was also a top priority.

Because, its target audience were the general public rather then an eclectic group of computer enthusiasts, Apple II sparked personal computer revolution. Thanks to the open design of the system that was extensively documented in the supplied manual, allowed people to write software and later design extensions for Apple II. At first simple games and software, that were written by other computer hobbyist, appeared in computer stores. As the popularity of the system grew, third parties designed different hardware extension cards. In 1978, Steve Wozniak in another feat of engineering ingenuity built an external $5^{1/4}$ -inch floppy disk drive, the Disk II, that attached via controller card to Apple II. The floppy drive took two weeks to design and build, just in time for the demonstration at the computer fair in Las Vegas along with newly designed office software VisiCalc. The introduction of floppy disk, which allowed for much faster load and store times, along with VisiCalc, opened business market for Apple II and launched it to new heights. Even though there were two other personal computers in the market, the abundance of features, great extensibility and later the myriad of software and hardware designed for Apple II, put it out of reach of its competition. Thanks to the great success of Apple II, Apple Computer Inc, went public in 1980.

Apple II was eventually superseded by Apple II Plus that was also designed around MOST 6502 microprocessor and had 48 KB of RAM, expandable to 64 KB. Apple II Plus included Applesoft BASIC programming language that supported floating point operations and as its predecessor Integer BASIC was stored in the ROM. Applesoft was supplied by up and coming software company Microsoft.

The Apple II Plus was followed in 1983 by the Apple IIe, a cost-reduced yet more powerful machine that used newer chips to reduce the component count and add new features, such as the display of upper and lowercase letters and a standard 64 KB of RAM, expandable to 128 KB. ProDOS operating system was introduced along with the machine. The IIe was the most popular Apple II ever built and was widely considered the "workhorse" of the line. It also has the distinction of being the longest-lived Apple computer of all time -- it was manufactured and sold with only minor changes for nearly eleven years.

Realizing the success and great revenue potential of personal computers IBM scrambled to respond. The response came in the form of IBM PC in August of 1981. IBM PC was a result of new approach to design cheap alternative to then dominant Apple II computer. Rather than going through usual IBM process, a special team of thirteen engineers was assembled with authority to bypass normal company procedures, in order to release a product to the market. The team designed IBM PC with off-the-shelve components. It was based on Intel 8088 microprocessor that run at 4.77 MHz, had 16 KB of RAM, expandable to 64 KB. The original PC had Microsoft BASIC loaded into ROM. It had Color Graphics Adapter video card that could use standard television for display. The standard storage device was cassette tape or a floppy drive for additional cost. No hard disk was originally available. IBM had an extensive penetration of the business market and decided to go with the open hardware specifications allowing for third parties to replicate the design. Even though, original IBM PC was too expensive for home users, it proved to be a great success with businesses. Eventually the IBM PC design spread, providing massive support for hardware and software. The consumers opted to join a larger world of PC clones which provided greater support. According to Steve Wozniak, Apple's consulting experts advised against allowing cloning of the system, since it was believed to be a mistake.

The operating system of choice for IBM PC and clones were versions of Disk Operating System (DOS). MS-DOS was single user, single task OS with command line interface and batch scripting facility. Originally written by Tim Paterson from Seattle Computer Products in 1980, it was first licensed and later bought outright by Microsoft for total sum of $75,000, prior to IBM PC official release. IBM supplied PC-DOS, a validated and packaged version of Microsoft-DOS (MS-DOS), with their PCs. Because of the great penetration of the home computer market by PC clones, MS-DOS managed to establish itself as dominant operating system.

Even though processor speeds progressed by leaps and bounds the RAM remained to be small and very expensive. The early models of personal computers were also lacking hard disks for permanent storage. As result the programs had to be optimized for size. The programming was done in high level languages, such as BASIC. But that required a relatively large memory footprint. The alternative of assembly language provided less portable but faster and smaller programs and thus was favored for system applications such as OS. Also, some business applications were written in assembly, such as Lotus 1-2-3. Because of its features and most importantly speed it became the killer-application for IBM PC

Personal Computers (1990s-2000s)

While Moore's law has been followed consistently through the years, 1990s saw an unprecedented growth in CPU clock rates. Semiconductor manufacturers were able to roughly double the clock rate every year, producing very powerful CPUs. Combination of very fast CPUs, growing amount of memory and disk space, and the advent of the internet – have created a booming demand for personal computers at homes and businesses, opening new markets for software makers.

Many agree that while the speed and size of various computer components has grown at astounding rate, applications developed for these systems do not benefit users by the same factor. By rough approximation, machines of early 2000s are about 10,000 times more powerful than average machines of 1970s; however the applications are not 10,000 more beneficial to its users. With such a discrepancy, one would think that applications running on such systems would hardly ever have any performance issues. As any hardware or software developer would knows this is not true. If applications are not 10,000 times more beneficial to users and still suffer from well known performance issues, then where does all the processing power go? Following analysis tries to answer this question:

1. With advances in hardware and operating systems support, today's PCs have many characteristics of supercomputers. Most modern operating systems support multiprogramming and multithreading, which benefit users by perceiving that many different tasks run at the same time. Unlike server systems, where it's

typical to have the entire server dedicated to running only one applications, user PCs can have many applications running at the same time, and competing for system resources. We see this trend becoming more and more prevalent, where with each new release of Windows we see more and more parallel services running together. While individual applications have only marginal benefits from increased power of PCs, users are able to work at much higher throughput because systems are able to run many simultaneous tasks.

2. Powerful PCs have lead to development of new code execution environments. Compilers, libraries, interop layers and managed code frameworks (such as .NET and JVM) allow application developers to be much more productive and cut development time by paying penalty during code execution time. Users typically are not concerned with technology used to implement applications. However, users are concerned with quality and release schedule, which are both improved with managed execution environment.

3. Current operating systems don't make full use of very large RAM sizes on PCs. Having a PC with a lot of physical memory is only useful when that memory is filled with useful code or data for applications that need it. Most modern operating systems still use old policies and algorithms for figuring out what pages need to be in memory and at what time. In the old model (where physical memory was very expensive), the operating system's job was to only maintain actively used pages in memory, while swapping any unused pages out to disk. As a result, many client applications don't fully benefit from having very large amounts of physical memory. New models are immerging where large parts of unoccupied memory are used as caches, which are filled with relevant code or data, based on some profiling data collected during normal computer usage. Windows Vista's SuperFetch does exactly this by recording usage patterns, and using those data to pre-fill memory with code and data that are most likely to be used in the near future.

4. While CPU clock speed has grown at a very rapid pace during 1990s-2000s, the speed of other system resources has not. Namely, disk and memory latency have not seen substantial growth since 1960s, and are major factors in limiting system performance in today's PCs. A typical user application is I/O bound as a whole, with small periods of time when it becomes memory or compute bound. Most performance architects (working on user applications) spend most of their time figuring out how to make applications less I/O bound. Techniques such as caching, working set reduction, profile-guided optimization, careful memory layout, and others are used.

5. In conclusion, another point should be considered. Is it reasonable to assume that new useful code for user applications can be developed at the same pace as 1990s-2000s phenomenal CPU clock growth? While I believe that disk and memory size increases do pace together with the rate of new software development, I contend that the CPU clock rate growth greatly outpaced it. If this is true, then

the answer to where all the CPU power goes is very simple – nowhere yet, because software industry has not yet caught up with high CPU clock rates (for user applications).

The point about user applications being mostly I/O bound should be further discussed. A natural question is why would a user application be I/O bound on a PC with 1GB of RAM?

1. When considering steady state, most user actions will work on code and data which are already in memory, making most user scenarios memory/compute-bound, giving users fast response time. While considering steady state makes sense for applications running on the server, client application performance is measured in terms of response time to user actions, which may or may not put the application out of the steady state. In many cases users launch other application features, read new data into an application, or simply run other applications in parallel. Any of these actions can put a given application out of steady state and cause a series of page faults, which in turn cause the response time to drastically increase for a period of time. Even though most of the time an application runs in a "warm" mode (where there are no hard page faults) giving user a fast response time, it is those infrequent situations when users experience slow response times which get attention of most performance developers and architects working on user applications. Because in most cases the slowdown in response time is attributed to not having the right set of pages in memory (and require I/O requests to satisfy hard page faults) – it is said that user applications are I/O bound.

2. Many user applications work on data coming from an I/O device (disk, network card, USB stick, etc), and are therefore bound by performance characteristics of a given device.

So why has disk and memory latency not experienced any significant improvements over past several decades?

Electronic storage devices have three main requirements – to be non-volatile, large, and cheap. While there have been efforts to develop non-volatile memory without involving mechanically moving parts, most of those efforts have not been able to meet the other 2 requirements (large and cheap) to be marketable. As a result, vast majority of large electronic storage devices continue to be hard disks. Disk latency is defined as the amount of time it takes for the selected sector to come around and be positioned under the disk read/write head. In an effort to address common issues present with hard disks, a new jointly developed (by Samsung and Microsoft) technology – hybrid drive – is coming to marked in 2007. Hybrid drive will offer up to 1GB buffer of non-volatile flash memory for caching most often used data. Besides decreasing latency (for certain data accesses) the flash memory buffer will also allow the platters of the drive to be at rest most of the time, compared to spinning most of the time with regular drivers. Less moving mechanical parts is expected to improve reliability as well as reduce power consumption.

“Memory wall” term has been coined in 1995 to refer to the growing discrepancy between CPU and memory speeds. It was approximated that between 1986 and 2000, the CPU speed improved at 55% per year, while the memory speed improved only at 10% per year. It’s important to note that as with any data transfer, memory speed consists of latency and throughput. While memory transfer throughput has seen improvements comparable to CPU, it’s memory latency which has been lagging behind. This issue was first identified by John Backus in 1977 as “Von Neumann bottleneck”. As CPUs becomes faster and memory becomes larger, the bottleneck becomes more and more a problem. Backus advocated a solution of using a mechanism where large amounts of data get read or written to/from memory, instead of “pushing vast numbers of words back and forth”. However, modern (object-oriented and functional) programming languages have not embraced this solution.

Today’s systems use several techniques for dealing with memory latency:

1. Reducing the latency. This is primarily done with CPU caches located on the same die with CPU. Getting the data out of the CPU cache has a significantly lower latency compared to main memory, which greatly benefits applications which are properly developed to take advantage special and temporal locality.

2. Tolerating the latency. One of the ways CPUs tolerate memory latency is by using hyper-threading (developed at Intel). A Hyper-threaded CPU simulates multiple logical CPUs using only one physical CPU. On a cache miss, the operating system schedules another thread to run on the CPU, which would otherwise be idle. The CPU also uses various techniques to prefetch data out of memory to prevent potential cache misses.

As with any market, economics has very big influence on how memory chips are made. Advanced circuit and layout techniques are potentially able to significantly reduce memory latency, but such techniques come at the expense of increased power consumption and cost. Low cost and storage size have always had very significant impact on memory design criteria.

The paper has discussed the CPU clock speed growth of 1990s through early 2000s. However, since 2004 semiconductor manufacturers have no longer been able to release chips with significantly faster clock rates. While Moore’s law for reducing feature size and increasing the number of transistors on a single chip continues to be followed, physical limits of semiconductor-based microelectronics present major challenges for increasing clock rates. Among those are heat dissipation, data synchronization, and others. Instead semiconductor manufacturers have began focusing on producing CPUs with multiple cores.

Before we get into how software is expected to benefit from multi-core CPUs, we’ll discuss why the approach of combining several cores on a single die makes sense, as opposed to other alternatives such as integrating more peripherals into the chip or significantly increasing CPU cache size). Systems with multiple CPUs have been manufactured since 1960s primarily for HPC and server market. An opportunity of placing multiple CPU cores on a single die presents several attractive advantages (over the traditional multi-chip approach):

1. Having CPU cores closer together reduces the distance electromagnetic signals have to travel between the CPUs to maintain cache coherency.

2. Multiple CPU cores laid out with proper geometry can use less power and space on printed circuit boards, compared to multiple CPU chips.

3. CPU cores can share common circuitry (such as L2 cache), which reduces cost.

4. Using the chip real estate for CPU cores is less risky or costly for semiconductor manufacturers than expanding the existing core with new functionality (such as integrating more peripherals). Also, significantly increasing CPU cache size in favor of multiple cores is impractical due to diminishing returns; while increasing the number of CPU cores (with right software) will continue to provide significant performance boost for systems.

Through early 2000s user applications automatically enjoyed improved performance due to rapid CPU clock growth. In fact, as we have already demonstrated, the industry used increasing CPU power to expedite development of new features and delivering products to marked faster. With stalled CPU clock rates, software industry has come to a point where existing and new user applications will no longer run faster with new multi-core CPUs. A dramatic shift in strategy, in order to have client applications scale with number of CPU cores, is currently a hot topic in the academia and the industry.

To make use of multiple CPU cores software running on the system must exploit parallelism. While HPC and server applications are typically developed using multiprogramming and multithreading to exploit parallelism, most user applications are not easily parallelizable nor have there been motivation for exploring on how to make them parallelizable. As a result many new research efforts are under way to explore various options of brining parallelism to user applications. Some of these efforts will be discussed further.

There is a very important dilemma, to which software industry is slowly waking up. It is clear that the model to develop new software features and expect the next wave of processors to automatically run the software faster will no longer work. So what model the software industry will adopt in the short and long term with respect to developing new software running on multi-core CPUs?

In the short term software vendors generally have 2 choices:

1. Invest substantially in adopting the model of writing concurrent software from HPC and server market. This would include completely re-architecting the product, training software developers to write concurrent code, and using different

development tools. With this model new features would exploit parallelism and would scale with the number of CPU cores.

2. Ignore the multi-core wave, and invest into optimizing the product for performance. As we have already seen, software industry used rapid CPU clock growth to accelerate delivery of products to market, making less investment in code efficiency and resource utilization. This leaves room for companies to make investments in using traditional methods of improving performance, and ensure that the product is optimized to maximize resource utilization.

Depending on the type of software, companies will generally adopt a short term model which falls somewhere between 1 and 2. A number of compute-bound parallelizable scenarios will be identified and implemented to execute concurrently. A good example of this approach is Excel 2007, where concurrency is used for executing several compute-bound classes of tasks. This leaves many user scenarios not using multiple cores. Because software vendors were not making large investments in performance or resource utilization in the past, I believe this leaves room for several years where additional investment in software can help run existing and new software faster. As discussed earlier, user applications are typically I/O bound, where CPU is idle for a large part of program lifetime. This leads to the point that disk and memory latency are still main bottlenecks in most user applications before the CPU is even considered. It should also be pointed out that GPUs are increasingly being used in today's systems for rendering graphics, letting the CPU execute application code.

While individual applications need to be changed in order to scale with multiple CPU cores, it's important to highlight scenarios where existing systems will immediately take advantage of multiple cores. Using virtual machines executing on a single physical machine will automatically benefit from having multiple cores, because each virtual machine can run concurrently independent of each other. Also systems with many processes executing concurrently will also benefit from multiple cores, which is inline with a trend in operating systems where more and more services run in the background.

In the long term, the industry is expecting new research to pave the way for collective development of new hardware and software to allow developing concurrent applications with little or no additional expertise on concurrency. While no single solution has yet to emerge in industry or academic research, there are several classes of efforts which are expected to drive the development of future hardware, operating system, and compiler features to ease the development of concurrent applications.

1. Many existing automatic vectorization and parallelization compiler techniques currently used exclusively in HPC market are being further researched and developed to enhance compilers used to compile user applications.

2. Techniques for augmenting code with attributes or hints to help the compiler make right decisions about concurrency have been around for many years. Namely OpenMP and MPI have become industry standards in this area. Similar

efforts are underway to extend popular programming languages with rich support for concurrent execution. Although this approach doesn’t achieve hiding concurrency from developers, many believe that this is the most realistic solution in the near future.

3. A completely new approach to solving typical concurrency problems (such as locks) is transactional processing. Researching this technique is currently gaining momentum from the industry and engineering departments of universities. Prototypes have emerged, and systems supporting some level of transactional processing are expected within next several years. Although researches have yet to agree on the right mix, transactional approach will require additional support from hardware and operation systems.

Bibliography.

Early computers:
"A Logical Coding System Applied to the ENIAC", R. F. Clippinger
"ENIAC" http://en.wikipedia.org/wiki/ENIAC

Early computers II:
http://www.ibm.com/
http://www-03.ibm.com/ibm/history/exhibits/701/701_intro.html
http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP704.html
http://www.ibiblio.org/pub/languages/fortran/ch1-1.html
http://inventors.about.com/library/weekly/aa072198.htm
http://en.wikipedia.org/wiki/Fortran
The FORTRAN Automatic Coding System for the IBM 704 EDPM, IBM October 15, 1956

Mainframes and minicomputers:
Interview: Joe Hellerstein (Microsoft)
http://www.cmg.org
"The Economic Value of Rapid Response Time", Walter J. Doherty (IBM), Ahrvind J. Thadani (IBM)
http://www.vm.ibm.com/devpages/jelliott/evrrt.html
http://en.wikipedia.org/wiki/OS/360
http://en.wikipedia.org/wiki/MVT

Personal Computers:
http://www.sasktelwebsite.net/jbayko/cpu1.html
http://www.intel.com
http://en.wikipedia.org/wiki/Apple_II
iWoz: From Computer Geek to Cult Icon: How I Invented the Personal Computer, Co-Founded Apple, and Had Fun Doing It by Steve Wozniak and Gina Smith

Interview: Hazim Shafi (Microsoft)
" Improving Performance in Excel 2007", Charles Williams (Decision Models Limited)
http://msdn2.microsoft.com/en-us/library/aa730921.aspx
http://www.dewassoc.com/performance/memory/memory_latency.htm
http://en.wikipedia.org/wiki/Hybrid_drive
http://en.wikipedia.org/wiki/Non-volatile_memory
http://en.wikipedia.org/wiki/Von_Neumann_architecture#Von_Neumann_bottleneck
http://en.wikipedia.org/wiki/Random_access_memory
http://en.wikipedia.org/wiki/Multi-core