# THE SEARCH FOR PERFORMANCE IN SCIENTIFIC PROCESSORS

JOHN COCKE

I am honored and grateful to have been selected to join the ranks of ACM Turing Award winners. I probably have spent too much of my life thinking about computers, but I do not regret it a bit. I was fortunate to enter the field of computing in its infancy and participate in its explosive growth. The rapid evolution of the underlying technologies in the past 30 years has not only provided an exciting environment, but has also presented a constant stream of intellectual challenges to those of us trying to harness this power and squeeze it to the last ounce. I hasten to say, especially to the younger members of the audience, there is no end in sight. As a matter of fact, I believe the next thirty years will be even more exciting and rich with challenges.

The three principal contributors to performance in scientific processors are the algorithm, the compiler, and the machine organization. When possible, the simultaneous optimization of these three factors holds the key to the highest possible performance. Of the three contributors, algorithm improvements are the most important. An idea that changes an algorithm from $N$**2 to $N$* log $N$ operations, where $N$ is proportionate to the number of input elements, is considerably more spectacular than an improvement in machine organization, where only a constant factor of run-time is achieved. Unfortunately, really exploiting the characteristics of a particular algorithm in the underlying machine organization often results in a specialized computer that will not perform well on most other algorithms. Signal processors are an example of this. They handle information transforms very fast, even in one clock cycle, but can be extremely inefficient in computing anything else.

My interests have been in achieving high performance for a broad range of scientific calculations. Therefore, I have focused mainly on optimizations involving the compiler and the underlying machine architecture. It is very probable that in the future, the highest possible performance improvements must also include the more difficult problem of algorithm modification. This is because parallelism will be essential, and the performance of parallel machines depends heavily on the algorithm used and how well it matches the parallel architecture.

In the last 25 years we have seen a 100-fold increase in uniprocessor computer performance. We have also seen a 10,000-fold decrease in cost. In the next 25 years, we are likely to see similar cost improvements. However, because subnanosecond cycles will be difficult, the performance of our largest machines will be on the order of a few billion instructions per second. We should always look at the absolute performance of uniprocessors, because if we can partition the problem, multiple processors may be used to gain even higher performance. We have seen this concept in trivial use for years, for example in job entry scheduling programs. Therefore, a key unresolved question is to understand how to partition a given problem at a global level and then operate on its separate parts using parallel machines. Global partitioning is a very difficult problem. Today there are many ideas, some good at specialized tasks, yet there is nothing promising for a uniform approach.

Today's compiler technology is quite sophisticated. Those who build machine architectures and organizations should design them so the known compiler optimization techniques are easily applied. New compiler concepts will be needed to exploit the capabilities of the new machines. I also believe that additional improvements are needed and are possible for existing optimizations. Better register allocation is one example.

The construction of a special purpose processor of very high performance is not difficult compared to a general purpose processor. By "special purpose," I mean concepts such as floating point units, caches, and vector processors. A machine built with a cache and a vector unit is specialized toward problems that have a good cache-hit ratio and are amenable to vectorization. It will have poor cost/performance on other problems. With relatively little progress on the partitioning problem and an apparent limit on the speed of a uniprocessor, we may find tomorrow's high performance world not much different than today's. Indeed, there will be a large class of problems that cannot be partitioned practically and thus, will be limited to uniprocessor performance.

I would like to describe to you the three most interesting projects that I have participated in. They were interesting because I felt I learned faster during these projects than at other times.

When I joined IBM in 1956, the Stretch project was underway. The project was led by Steve Dunwell, who placed performance, not cost, as paramount. My experience had consisted of writing one Monte Carlo simulation where the machine language was HEX. I am shocked that I did not write an assembler before proceeding, but the problem was so interesting that such a thing never occurred to me.

Designed for Los Alamos, Stretch's ambitious goal was to be 100 times faster than the existing 704 while providing great flexibility in addressing, floating-point arithmetic, and nonnumeric operations. Any bit in the machine could be addressed directly, any word could be monitored, variable length data could be referenced, and floating point numbers came in many varieties. It was a programmer's dream—especially assembly language level programmers—and it was a wonderful challenge for those of us designing the hardware.

To overlap memory access, we introduced instruction execution look ahead (pipelining). Error Correcting Code (ECC), was applied to main memory and disk and tape I/O, and we worked with the compiler people to develop efficient instruction sequencing and register allocation.

Fortran I for the 704 generated excellent code, even measured by today's criteria. Those interested in compilers learned a great deal from Fortran's usage with Stretch. For example, much of the richness of the Stretch architecture was not exploited by the compiler. As I recall, John Backus told us in advance that this would be the case. We architects would have benefited had we known more about the Fortran compiler at the time. Because of this and many other experiences, however, we have learned that it is easier to write compilers that capitalize on a simpler instruction set. Although Stretch met less than half of its performance goal when it was shipped to Los Alamos in 1961, we had invented and tried many techniques still used today.

Next, I would like to tell you a little bit about the Advanced Computer System (ACS). This was a project we undertook between 1964 and 1968. It had a simple yet irresistible goal: to design and build the fastest scientific computer feasible. Under the late Jack Bertram's leadership, we designed a computer with many organizational features that even today are not well known.

For various reasons, however, ACS was not built. It would have had a ten nanosecond cycle time, on the order of today's multimillion dollar computers, so it would have been very fast hardware. This is only where machine organization begins. At each cycle we dispatched seven operations, one to the branch unit, three to the fixed point unit, and three to the floating point unit. The fixed point unit could initiate three instructions on each cycle. The floating point unit had a buffer of eight operands and logic to pick the first three out of the eight that were ready. We had two paths to cache allowing two memory accesses per cycle, and to match the cache access time to the CPU cycle, it was pipelined five deep (necessary because of the low level of circuit integration). A neat invention was a FIFO store queue that allowed computation to proceed without waiting for a store-through and a hardware interlock to protect the cases when a reference is made to the operand. It also resolved interfering load instructions.

Much of our effort went into dealing with branch instructions in ways that minimized draining the pipeline. We divided each branching instruction into its three essential operations: determination of condition, calculation of branch target address, and the actual jump instruction. There was also a branch history table that allowed instruction prefetching based on the dynamics of recent branch instruction executions. Another specialized form of branch was called "skip;" it allowed compiler scheduling across branches. Given all of these hardware assists, the experimental compiler generated good code and scheduled instructions to minimize the pipeline effects.

Long before we had a firm hardware design, we had an experimental optimizing compiler to evaluate variants on the design. Working with a combined team of compiler and hardware people, I learned the importance of not including hardware features that the compiler could not use and including hardware facilities to allow efficient compilation. This experimental compiler was the source for much of our subsequent work on optimization algorithms. Many of the code optimization methods used in compilers today came from this work. These include interval-based control flow analysis, data flow analysis, common subexpression elimination, code motion, strength reduction, and code scheduling. On occasion, the compiler was able to generate better code for ACS than the best hand coders. We learned of the importance of an efficient compiler and its surrounding software support tools.

The parts of ACS that were built 20 years ago are still impressive, except for the speed-power product: internal circuits were 250 picoseconds, each circuit dissipated about 30 milliwatts, and there were up to 40 circuits per chip. Had the complete machine been built, I believe that the cycle time would have been in the low teens of nanoseconds. It would have accomplished four to five instructions per cycle on linear algebra-type problems. But, because of cache latency and the inability to almost always anticipate the correct branch flow, we would not have achieved near that rate on more general purpose problems.

*Another important part of the 801 project was the tight coupling and simultaneous development of the hardware and the compiler.*

These are only a few of the features we had in ACS. Many ideas have found their way into subsequent machines at IBM, particularly in the cache area. Some of the more intricate ones, however, have not yet been used. In many cases this was due to cost versus performance trade-offs, but I am confident that declining costs will enable use of these ideas.

ACS never made it out of the laboratory; I suppose it was too big and too expensive, but for me it was probably the most exciting project I have ever been involved in. In reflecting on this, I believe that what made it particularly exciting was that we were a small team, mostly hand-picked by Jack Bertram, and we pioneered every aspect of the project.

I spoke about the architecture and the compiler, but some of our most clever inventions had to do with testing instruments and techniques, fast disks, packaging, and cooling. I received a great deal of satisfaction in watching our engineers tackle and solve these problems. Another principal factor was the quality of the team. Many are names I am sure you will recognize—among them Fran Allen, Dick Arnold, Fred Buelow, Phil Dauber, John Earle, Charlie Freiman, Russ Robelen, Herb Schorr, and Ed Sussenguth. My only regret about ACS is that unlike compiler ideas, we did not take the time to publish our ideas on hardware so others could build on them.

Let me now move on to discuss another interesting project I was involved in—the 801 computer. Around 1974, we were investigating the possibility of building an all digital telephone exchange capable of handling approximately one million calls per hour. At approximately 20,000 instructions per call setup, we calculated that we needed a processor capable of executing 6 million instructions per second (MIPS). But since we did not know that much about telephones, we felt that we should probably aim for a 12 MIPS processor. Certainly, a general purpose machine was not the best choice for this task. For example, we had stringent real-time response requirements. So in a sense, we were looking at a special purpose machine. From the beginning, we also assumed that we would program in a high-level language. In this case it was not absolute performance that motivated us. We had a performance target. We knew the general nature of the applications; we had no heavy floating point calculations, and we were looking for the lowest cost computer that would meet these requirements. We designed a machine that subsequently became known as the 801 Computer. Eventually, we abandoned the idea of a telephone exchange, and concluded that we had pretty powerful ideas in computer design. We complemented these ideas with a high level language and a compiler, PL.8, and pursued their development for the next several years.

Among the principal features of the 801 were separate instruction and data caches, providing much higher bandwidth between the memory and CPU, and no arithmetic operations to storage, thereby greatly simplifying pipelining. Another important part of the 801 project was the tight coupling and simultaneous development of the hardware and the compiler. This tight coupling allowed the construction of an effective compiler and provided a simple machine organization.

This project was a great team effort, and my colleagues Marc Auslander, Greg Chaitin, Al Chang, Marty Hopkins, Peter Markstein, and George Radin, to name just a few, were not only great contributors, but made the whole venture enjoyable. Another satisfying aspect of the 801 project was that many of our ideas were considered sufficiently interesting by others and stimulated considerable additional research and experimentation in many universities. Berkeley coined the name "RISC" (Reduced Instruction Set Computing) for similar work.

Within IBM we feel it is extremely important to simulate the hardware of a new machine extensively. As the complexity of machines increased, the time required to simulate them was becoming too long. In 1980 we started to develop a special purpose machine for this task. Rick Malm developed an early prototype and Monty Denneau designed, built, and debugged a very large system. Its goal was to provide a hardware assist to the logic design simulation, increasing the speed of simulation 100-fold to 1000-fold, depending on the number of circuits to be simulated. Logic simulation appears to be an ideal problem for parallel machines because it should simply emulate the way a computer really works. The machine has 256 parallel logic units accessing a shared memory. A special compiler takes a reasonably high-level logic description, generates the necessary instructions, and loads them into the machine. Each unit simulates about 4000 circuits sequentially and at the end of each cycle, broadcasts its results through a nonblocking switch to each of its 255 partners. Thus, 256 circuits per cycle are simulated.

Assigning logic blocks to a machine is analogous to assigning circuits to chips, and according to Rent's Law, 4000 I/Os for 4000 circuits should be more than enough. In spite of this, the compiler partitioning problem takes a very long time on a large mainframe CPU. This forces us to use the machines only for long-running simulations. In spite of this time-consuming difficulty, many such machines are in constant use at IBM. The next generation logic simulator, with added switching capacity, is being built to ease this problem.

Note, however, that the nonblocking switches grow at least as fast as $N^* \log N$, whereas the machine hardware grows linearly. Thus, in the limit, it will be *all-switch.* I have mentioned this example because it demonstrates a paradox simply: A priori, it seems very natural and simple to map a logic simulator onto multiple machines, but a simplistic straightforward extension will encounter difficulties.

*It is this trend that emphasizes cost over performance that leads me to believe the search for future scientific computing performance has to concentrate on gross parallelism.*

This brings me to the future. By exploiting improvements in circuit and memory density (e.g., one million transistor chips and four megabit chips), it is possible with just a few CMOS chips to build a powerful scientific machine that will match the performance of vector based computers on many important problems. This machine will have a very fast floating point multiply/add unit equipped with buffers on a single chip. It also will be capable of simultaneously executing a fixed point, a floating point, and a branch instruction. As has been stated several times, optimizing compilers will be essential to exploit such machines.

Memory is inexpensive and cost improvements will continue. Thus, we can expect large random-access memories consisting of hundreds of gigabytes. Disks will maintain the backup database, but by paging, the database applications will execute out of main memory at speeds substantially faster than they do today. The significance of this is that the enormous amount of effort that has gone into developing schemes to achieve high performance on mechanical devices will no longer be necessary.

As I said before, I expect the thirty-year-old trend of 100-fold computer performance improvement with 10,000-fold cost improvement to continue. It is this trend, that emphasizes cost over performance, that leads me to believe the search for future scientific computing performance has to concentrate on gross parallelism. It is necessary due to increasing difficulties in reducing cycle time and cycles per instruction. Due to the continuing steep decrease in costs, it will be possible to aggregate many CPUs executing multiple instruction streams concurrently. Thus, one of the principal challenges will lie in compiler optimization techniques, both to recognize the parallelism and schedule the instructions. One particularly difficult yet crucial problem is to minimize cache misses in these parallel processors. For example, on an inner loop that goes at three instructions per cycle on a machine that has a cache taking 10 cycles per miss, a 3 percent miss ratio will halve performance.

Thus, machines such as this can be considered specialized in the sense that problems with good cache hit ratios achieve high performance. I see many techniques where tricky programming can vastly improve the cache hit ratio. I believe that people interested in compilers should try to make these tricky techniques automatic, as vectorization has been made automatic by compilers.

I mentioned earlier that improved algorithms can provide the most leverage, orders of magnitude in the dimension of the problem, whereas parallelism will at most provide a speedup linear to the number of processors. New algorithms tailored to such parallel machines that exploit the clever details of their architecture and interconnection will be essential, and conversely, specialized machines oriented towards such parallel algorithms will be possible and economically warranted.

Many years ago, John McCarthy remarked to me that if it had not been for Alan Turing and his idea of a Universal Machine, we would still be arguing about the capabilities of different machine designs. Fortunately, this is not an issue, however, we do seem to carry on at length about cost/performance of various machine designs. The arguments are further complicated since each machine can demonstrate a set of problems and algorithms for which it is *specialized* and does particularly well. I am certain this will be worse for parallel machines where the degree of specialization can be higher. Comparison of various machine architectures is not done well today, perhaps because of its intrinsic difficulty or perhaps because of commercial implications. I hope this can be done more carefully in the future; maybe we could look to academia for help.

I would like to conclude by reiterating that the past thirty years have emphasized cost improvement over performance. I see no reason why this trend should not continue. As we look ahead, we see ourselves approaching a limit on the performance of a uniprocessor and therefore, we observe many working on multiple machine aggregates. As it is not obvious that this will solve the high performance scientific computing problem, we should not infer that the future is simply an extrapolation of today's ideas. I do not find it discouraging that there seems to be no clear-cut route to high performance. I feel the flexibility of computers will allow us to solve problems in ways not yet envisioned, and will make the future of computing more interesting than the past.