What Computers Can't Do (And Why)

Now that computing is approximately 30 years old we are in a position to look back at the progress that has been made to the general state of computing as seen by the public and by advanced workers in computing. How much have we advanced? How much better are our facilities now than 30 years ago?

Not much.

There have been many improvements in both hardware and software; computers have become affordable by home enthusiasts; most companies use them for daily activities. But uniformly the progress made in computing have been akin to the improvements that were made in horse-and-buggy technology over the centuries—impressive but nothing revolutionary. Nothing in computing rivals the leap that the internal combustion engine represented to transportation.

1. What Computers Ought To Be Able To Do

Almost every person in the US can walk up to a telephone and know how to use it; this is not true of computers. There is variability in keyboards, operations systems, command languages, mouse control, and peripherals. Therefore computers, which are used daily by a large fraction of the population, are not as commodious as telephones. The fault lies with the lack of advances in computing that we have seen since computers came into existence 30 years ago.

Here are some examples of the things that computers ought to be able to do but cannot.

A computer is not aware of what you and it are doing. I don't mean this in the sense of artificial intelligence, but in the simple sense of knowing that the current task, for example, is to write a document using a text editor, a typsetting program, and a laser writer. If there is a set of typesetting macros I use but which I have not used for several months, the computer should know the name of the file containing them. If I walk away from my computer for a couple of hours, it should be able to remind me what I was doing. It could tell me I was editing a file, running a typsetter over it, and printing the results.

If I have a special telephone, I can carry it with me and plug it into any wall socket and use it. If I have a favorite way of interacting with a computer or some programs for it, I cannot easily transfer my personal user interface to it. In fact, I cannot usually customize the interface to a general program. For example if I want to use the text editing commands

of my favorite editor to move around the cells of a spreadsheet, I cannot easily accomplish that unless the authors of the spreadsheet have forseen my (and others') desire to do so.

People have been writing programs for 30 years. There may be a program that I want to construct that comprises three existing programs and a little bit of code to put them together. If the source code does not still exist, I cannot easily accomplish this; and even if the source still exists, I might have trouble because the languages are different from each other or different from what they were when the programs were written.

I cannot use parts of these already-existing programs in new programs I want to write. I can either run the entire program or try to duplicate the fraction of the behavior I want.

The operating system contains lots of interesting and useful pieces of code. For example, the operating system can create an address space and run a program in it. I cannot write a program that invokes that part of the operating system—that is, I cannot write a program that creates a new address space, runs a program in it, and gathers the results.

I cannot easily write or put together multilingual programs. If I want to have a program with part written in Lisp, part in FORTRAN, part in COBOL, and part in APL, I am generally out of luck.

The key to understanding the shortcoming of current computers is to forget about the phrase "writing a program" and concentrate on the phrase "construct a system." If I want to have a stereo system with a particular turntable, CD player, preamplifier, amplifier, speakers, and tuner—where my preferences are based on performance, specifications, musicalness—I don't buy the transistors, chips, op amps, power supplies, and the rest of it and put the system together; rather, I buy the already-constructed large components and cable them together. I am able to build a respectable sound system using my knowledge of cabling and my good taste and sense; I am not required to have a knowledge of electronics.

Similarly, I should be able to build a large system out of large already-constructed programs, using a knowledge of "cabling" them together and my good taste and sense. But, I cannot routinely do that today.

The uses of computers that have been exploited by computer scientists so far are: 1) running existing programs; 2) writing programs in a single programming language. Programming requires a detailed knowledge of a programming language along with a fairly high degree of mathematical-like talent.

Early cameras required a lot of detailed knowledge and talent. One had to know about chemistry, needed to know how to use sophisticated light-measuring devices, and one had to use a finely-honed talent to trade off shutter speed and aperture. Excellent photographs could be taken by only skilled artists. Now that camera designers had concentrated on making beautiful photography easy, ordinary people can taken excellent pictures whose beauty is limited by only the photographers taste and artistic ability.

Why can't there be an analagous story for computers?

2. The Act of Programming

The act of programming, one of the few things computers help me do, is not especially well facilitated. When I am programming, I am creating pieces of code that I want to try out along with some data structures. Unless I am writing a simple algorithm, well-thought-out and specified to begin with, I am exploring computational behavior. Sure, textbooks tell us that programming is the design of crafted algorithms, and some of my programming is like that, but most of it is like painting—I put code and data structures on the computational canvas and I stand back to see how it performs.

When painting, the artist does not need to start the canvas over when he makes a mistake or doesn't like a section—he can remove or cover the bad spot. Why shouldn't programming be like this? Is there some virtue to doing things the hard way?

3. Operating Systems

Let's look at some of the high-visibility areas in computers and see how well the current state-of-the-art, as seen by real users, measures up to what 30 years of work should have produced.

Consider operating systems. The currently available and widely-used ones are IBM MVS, MS-DOS, Unix, Macintosh, and Vax VMS.

IBM MVS is a 25-year-old operating system. It was designed to enable many people to use a normally batch-oriented computer system.

MS-DOS is based on 30-year-old technology. The fact that people refer to it as an operating system is remarkable, because it is little more than a set of device drivers and a command processor.

Unix is a 20-year-old operating system, designed to run on the PDP-7 (?). After its initial implementation it had timesharing capabilities grafted on top of it. Its most

advanced feature is the ability to "pipe" an output stream of one program to the input stream of another. Although this is a nice feature, no generalization or extension to it has been made in 20 years.

The Macintosh operating system is the most advanced of the lot. It provides a uniform user interface capability that is easy for the applications programmer to use. On the other hand, the operating system does not support virtual memory nor does it allow multiple programs to easily work together. The fundamental ideas are based on those found in SmallTalk, which was developed 15 years ago.

Vax VMS is a PDP-11 operating system derivative. Again, it is 20-year-old technology.

4. Traditional Programming Languages

Today, the prevalent traditional programming languages are FORTRAN, Pascal, and C.

FORTRAN is 30 years old. The name stands for **FOR**mula **TRAN**slation. Early high-level programming languages were little more than front-ends to assemblers, and FORTRAN was about as simple a front-end as could be imagined in the late 1950's. Over the years FORTRAN has become more sophisticated, but it is still remarkably similar to its progenitor.

Pascal, a younger language, is in the spirit of Algol60, which was defined more than 25 years ago. Pascal, along with C, is in a class of programming languages that are designed to help a compiler produce small, fast, compactly-represented code. Whatever convenience is provided is provided to the compiler-, loader- and operating-system-writer rather than to the programmer.

5. Innovative Programming Languages

The traditional programming languages are distinct from innovative programming languages in that traditional languages provide the programmer a means for producing code that accomplishes precisely the objectives apparent in the code while innovative languages provide the programmer a means for producing code that resides within an interactive programming enviroment and can be altered, traced, encapsulated, and built upon. In addition, a traditional programming language is based primarily on the model of a main program calling subroutines to accomplish tasks. The innovative programming languages are designed to give the programmer a different model of code objects.

The three prevalent innovative programming languages are Lisp, PROLOG, and SmallTalk.

Lisp is 30 years old. The code objects in Lisp are functions, which are called to supply values to other computations—Lisp programs are functional compositions. Lisp programs are intended to be run within an interactive environment.

PROLOG is 15 years old, and the code objects in PROLOG are Horn clauses. Programming in PROLOG is defining a series of logical statements about the relationships among a set of objects, and program execution is the chaining together of these statements. Additionally these statements can side-effect a database of facts. Like Lisp code, PROLOG code is executed within an interactive environment.

SmallTalk is 15 years old, and code objects in SmallTalk are methods associated with classes. Objects in Smalltalk are instances of classes, and action is taken by sending an instance a message, which is handled by the appropriate method. Again, this takes place within an interactive environment.

SmallTalk is an object-oriented language, PROLOG is a logic language, and Lisp is a functional language.

One thing to notice about these innovative languages is that they are structured very much like a language and an operating system combined. For example, when writing Lisp code, an environmentally-resident compiler produces code that can be executed without a loading phase. Memory is managed by the Lisp system, and advanced Lisp systems have a timesharing capability.

A traditional language requires a compiler as a separate program, a loader, and an operating system within which to run the compiled code. One could say that the functionality of the innovative programming languages is unbundled in the traditional ones.

6. Why Are Computers So Bad?

The primary reason that computers are so bad is that software technology is backward. It doesn't help that almost all computers are designed to run compiled FORTRAN or C.

Software is backward because programming languages, programming environments, and program environments are backward, and these are backward because there are only two major concerns for software: speed and small size.

7. Size

Computers were once small, having very little physical memory, and virtual memory did not exist. The early traditional programming languages were designed for writing code to run in these small computers, and the early compilers were designed to run in them. Therefore, the programming languages were designed, implicitly, to write small programs, and the compilers had to be simple and small. Because the compilers had to be simple, the language had to provide means for the program writer to tell the compiler everything it needed to know.

Programming methodology was geared towards small machines—people dwelled on how to produce small correct programs. Programming in-the-small was the only possible concern, and people were (and are) concerned with it.

Early operating systems were designed to be small and out of the way of the user's program. Therefore, operating systems enjoyed minimal functionality, they relied on heavy re-use of machine language code, and their facilities could not be exported to user programs.

8. Speed

Early computers were slow; they were measured in thousands of instructions per second. Memory systems were slow; disk systems were small and slow.

High-level programming languages (like FORTRAN) had to produce very efficient code to enable programmers to write programs that accomplished reasonable tasks. Programming language design focussed on how to get the most out of a computer. The acceptability of a proposed programming language feature was measured by how well a compiler could do with it and how many instructions it required.

Language design included means for telling the compiler about the types of objects and the layout of data structures. A type system was considered useful for a programmer and for a compiler, but not for running code. Early compilers were little more than a preprocessor for assemblers, and because programmers could neither take the time to arrange for runtime types nor afford the extra space for them in assembly language code, neither could the compilers nor could the compiler writers.

Any sort of memory management was considered harmful except for stack discipline—allocating objects on a stack so that the objects disappeared when the stack boundary moved past them. Smart, flexible memory management requires that data structures have

explicit types, and this was considered an extravagance for the computer, though people required types to help with programming.

Operating systems provide some runtime support for running programs, mostly by providing input/output capabilities, address-space management, and miscellaneous functionality. Because computers were slow, the operating system had to be very efficient to stay out of the way of user programs. Therefore, the most important design consideration next to size was speed. Is it any wonder that the early operating systems and their kindred spirits in use today have little functionality?

9. Programming Languages

As we saw, current programming languages either are old or are simple derivatives of old programming languages. These old languages were designed to produce small fast code with the burden of effort resting on the shoulders of the programmer.

In addition, these languages were designed in an era when the prevailing attitude was that programs were to be written once and used very many times, by many people. Because this belief molded efforts of language designers, it is not surprising to see the resulting languages used exactly this way—the path of least resistance is followed.

10. Traditions

Traditions are important in both culture and science. Tradition controls what we can believe, what we can do, and the thoughts we can have. In science, it controls the kind of work we can do. In computing it controls the kinds computers, the kinds of operating systems, and the kinds of programming languages we can have.

Because the tradition of small and fast has been strong for the last 30 years, the computers, operating systems, programming languages, and programming methodologies we have are very much like the ones we had 30 years ago.

11. Cost

There are many things that have changed in the last 30 years that invalidate the assumptions made by early computer scientists. Now computers neither are slow nor are they small.

More interestingly, they are no longer expensive.

In 1961 the average programmer salary in the Boston area was $5,000 per year. In 1961 Texas Instruments introduced the first hand-held calculator, which had 5 functions; it cost $29,000. Today the equivalent programmer salary is $45,000. If the ratio of people cost to calculator cost that held in 1961 were used to figure a minimal computing cost today, each programmer would have $261,000 of computer.

In 1961 computers were very expensive in relation to people. Therefore it made sense to spend a lot of effort to make sure that the computers were well-used. In fact, a hand-held calculator was equivalent in cost to nearly 6 programmers, who could be kept busy programming it for best efficiency. The quest for small and fast in 1961 was justified.

Today it isn't. Benchmark programmers, not programs.

12. Hardware versus Software

Now that we've looked at some of the technological and tradition-based reasons for the sad state of computing, let's look at some business reasons.

Twenty to thirty years ago the typical hardware company years ago was primarily concerned with selling iron. Companies designed complex and expensive computers, and they wanted to expend the least amount of time and money in making them usable. Furthermore, the hardware companies wanted to show off their hardware in the best light, which meant small, fast operating systems that were good enough to do the job. One might say that a hardware manufacturer treats software the way a pimp treats a miniskirt—a decoration to improve the seductiveness of the wares.

The companies did not want the cost of software to hamper the hardware sale, so they either priced the software low or they gave it away. Today it is common to see hardware manufacturers giving away operating systems, compilers, and many utilities.

Even a company that wanted to charge for software was in a quandry as to how to price it. Software is not tangible in the same way that a computer is: You cannot trip over it, you cannot weigh it, but you can accidentally erase it.

The situation for software authors is similar to that of book authors—book authors produce the words and stories and ideas that sell books, but the folks who provide the paper, ink, advertising, and distribution make the lion's share of the money.

If the hardware manufacturers have no incentive to improve software technology—almost by tacit agreement—then it is up to the independent software vendors. But these

developers are competing against the hardware companies, who have already devalued software. Because they compete against the hardware companies and against each other, they must offer some edge with respect to their competitors: Either they can lower prices or offer increased functionality. The latter is a pure risk while the former is a well-understood business situation. What we see is exactly what is expected: Most companies compete using price, and some make modest technological advances.

13. Can the Situation be Improved?

It is possible to vastly improve the quality of computing. It can be done with operating-system-level changes, with programming language changes, and, later, with hardware changes.

Most improvements require software and not hardware. The techniques for accomplishing our goals are well-known. The problem is that it costs speed and size to existing operating systems and programming technology.

What technolgy am I talking about? The innovative programming languages come very close to meeting the needs of our blue-sky system. These languages allow us to have interactive environments, dynamic binding, incremental developement, operating-system-like capabilities, and so on. They go only a fraction of the distance we need to go, but they lead the way.

The software technology we need costs speed and size. When computers were slow, small, and expensive, these costs were intolerable; now they aren't.

Consider motorized vehicles. Among them, the highest accelaration is found in motorcycles or dragsters; the fastest are Indianapolis cars; the largest capacity are 18-wheelers; the most fuel effiecient have insulated engines. But people buy and use Chevys, Toyotas, pickup trucks, and trail bikes. To the specialist these vehicles are not interesting, but to ordinary people they are.

14. Conclusions

When a field is new there are no traditions or constraints, and people are free to try all manner of possibilities. The innovative programming languages are about as old as the traditional ones. This is because there was no reason to not invent these languages. But economics and happenstance forced us in one particular direction.

Let's change direction.