# The Turning of the Wheel

**Neville Holmes,** University of Tasmania

Computing has, according to some recent popular articles, found something wonderfully new—*virtualization*.

Reading some of these articles, two thoughts struck me. First, that the word must be one of the ugliest and most awkward to be introduced recently and, second, that virtuality in digital technology is far from new.

Indeed, if we take written language as the second great digital technology, as we should, the scribes in the scriptoria of old Europe were virtual authors. More recently, in the early 1970s I worked interactively through the Cambridge Monitor System interface on a virtual System/360 computer provided by a hypervisor called CP67.

Three sources spurred me to draft this column: an April *Computer* article titled "Overcoming the Internet Impasse through Virtualization" (T. Anderson et al., pp. 34-41), a column in the same issue on the oxymoronic topic *virtual reality*, and *Computer*'s entire May issue dedicated to virtuality but with only high-level consideration of its history.

**The new in digital technology is not what we do, only what we do it with.**

## TIME SHARING

In the 1960s, computing manufacturers offered two kinds of machines:

- commercial, usually with decimal arithmetic; and
- scientific, usually with binary arithmetic.

Early digital computers had operators who ran individual programs when the programmers didn't. Given the machines' great expense, developers sought ways to automate their operation and increase their throughput.

These early methods combined job stacking—the automatic transfer of control from one program to the next with peripheral transcription—and the use of a cheap machine to transfer data between the fast magnetic tape used by the mainframe and the slow punched card and paper-tape machines and printers.

These batching methods reached their culmination in the mid-1960s with the somewhat chaotic but eventually successful introduction of IBM's System/360 operating systems, which used multiprogramming and SPOOLing (Simultaneous Peripheral Operations OnLine).

Because the 360 architecture combined binary and decimal arithmetic, IBM planners had imagined that their product would be as suitable for the scientific world as for the commercial. This prediction proved wildly inaccurate.

In the business world, management could dictate that users be kept at arm's length and programmers be banned from machine rooms, but in the scientific world, users ruled the roost. Users are more interested in getting good results than in keeping costs down. Scientific users, who often programmed for themselves, found the extra control in using the machines attractive. This led to the idea of time-sharing.

In its basic form, time-sharing relied on most users being at their Teletype terminals mulling over what happened last and what to do next, so that a ready-to-run user's program could take over the machine temporarily. In practice, time-sharing required virtual memory to be successful.

By the time IBM finally got its 360 batch operating systems up and running, time-sharing had established itself, particularly in universities, and it looked as if the company would lose a huge market. In response, IBM mounted two massive projects, one in Poughkeepsie and another in Mohansic.

The Poughkeepsie project, Time-Sharing Option, aimed to provide time-sharing as a subsystem of the top-of-the-line MVT operating system. TSO distracted developers greatly from the basic improvements that system needed, got off the ground slowly, and was not very successful.

The other project, a time-sharing system called TSS/360, built on IBM's experience in collaborating with universities on their time-sharing projects. IBM intended this system to run on a special 360, the Model 67.

TSS proved a complete flop. Many 360/67 users outside IBM switched to the Michigan Terminal System. When I joined IBM Australia's Systems Development Institute in late 1970, the

*Continued on page 98*

*Continued from page 100*

Institute was abandoning TSS for CP67/CMS. This is where the virtual machines come in.

## VIRTUAL MACHINES

Tom Van Vleck briefly describes the CP67/CMS's development (www.multicians.org/thvv/360-67.html), while Melinda Varian does so in delightful detail (pucc.princeton.edu/~melinda/25paper.pdf).

Early time-sharing adopted the idea of providing concurrent constrained use of the computer by users' programs.

A relatively small team at IBM's Cambridge Scientific Center in Massachusetts covertly implemented the forerunner of CP67/CMS, CP-40/CMS on a modified 360/40. They used the term *virtual machine*, having heard of it being used for an earlier, more conventional, IBM time-sharing system.

Their elegant work, starting in the last week of 1964, was inspired by the idea that they would provide for each user a strictly virtual machine indistinguishable from a real one by a user program. After the 360/67 intended for TSS/360 was announced, they converted their work to run on that machine.

The strict virtual machine had many advantages. The hypervisor or CP turned out to be relatively simple to do. Because all time-sharing users had their own "machines," with their own disk partitions, the CMS only had to support a single user and had simple support requirements compared to the requirements for conventional time-sharing.

Because the virtual machine was strict, we ran the ordinary OS/360 in a virtual machine for our batch work in Canberra. We even tested new versions of CP in a virtual machine.

Several observations spring from this earlier development. First, the term virtual machine had a specific meaning, and its use today is degenerate. Of course, all computing is virtual, but we should use technological terminology to enhance meaning, not remove it. For example, the Java virtual machine is properly a simulator, or, if done in hardware, an emulator. The use of emulation ensured the success of the System/360 machines.

More generally, the epidemic use of the adjective *virtual* is akin to the epidemic use of *user friendly* in the 1970s. Further, it's a pity that the pathetic initialism VMM has been adopted for what had once been more expressively and comfortably called a *hypervisor*.

**The epidemic use of the adjective *virtual* is akin to the epidemic use of *user friendly* in the 1970s.**

Second, in a large company, people make technical decisions for political reasons. Cloistered development managers in IBM rejected the virtual machine and sought to get rid of it long after it had saved the day for them. The computer industry today would have been quite different if IBM managers had enthusiastically adopted the principle once circumstance forced it on them.

## MULTICORE CHIPS

It's probably simple fashion that couples multicore chips with support for virtual machines ("Chip Makers Turn to Multicore Processors," D. Geer, *Computer*, May 2005, pp. 11-13). Certainly, strict virtual machines need hardware support, but multiprocessing has nothing directly to do with that.

Multicore chip development seems to have happened because the manufacturers of these chips have run out of ideas for using all the circuitry their improved manufacturing methods have made available. I would rather they had provided improved interval and complex arithmetic, or support for console windowing independently of the operating system. This would help foster the reportedly growing movement back to thin clients, known as dumb terminals in the 1980s.

Multicore chip development has two interesting aspects. First, consider the licensing issue mentioned in the *Computer* news item. If a chip has two processors using proprietary software, how many license fees must be paid? Second, continuing development raises a question: If multiprocessor chips succeed in the market, what then? More cores per chip seem likely.

How then will software adapt to these changing architectures? Another wheel could come full circle, one called *strict virtual architecture*.

## VIRTUAL ARCHITECTURE

A little-known wild-duck IBM project illustrates what I mean by strict virtual architecture: the System/38.

Universality was the official marketing story for the IBM/360: one architecture to rule the world, with the 360 being the number of degrees to a full circle. But IBM soon branched out into a spectrum of incompatible architectures—Series/1, System/3, System/7, System/32, and System/34, for example. Developers typically used these architectures for problems and customers too small to warrant a System/360 with its accompanying data processing department.

Eventually, IBM started a project in Rochester, Minnesota, to bring the small commercial machine architectures together. Their System/38 virtual architecture can best be described as glorious. Instruction addresses referred to objects so that, for example, the program had only two add instructions, one with two addresses and one with three. The three-address add could, for example, add a packed decimal value to a binary value and produce a character result. All objects were stored in a 64-bit address space, even though early machines used only 48 bits.

Disk storage supported the address space but could not be directly used from programs. For a long time, IBM supported only Cobol and RPG, but no assembler. Their code was compiled to instructions in the virtual architecture. The objects the compilers pro-

duced could not be run directly, however. The operating system used a machine instruction to convert a compiled object to a runnable object, at which point the virtuality appeared.

Creating the runnable program involved a translation from the virtual instruction set into the actual instruction set. This translation made the highly sophisticated object-oriented virtual instruction set possible, and this set hid the complexities of the actual machine from the programmer.

If an improved actual machine was required, a new *create program* instruction could be written. Making old programs run on the new machine would require only the creation of a new runnable program from the old compiled program objects.

All of which makes it seem that the best way to cope with the developing multicore chips would be to have a strict virtual architecure support them. This would hide chip complexity and changes from the compilers and interpreters.

Computing often recycles the old as new, and this holds true for strict virtual machines. Even the 360 name once used by IBM has recently been recycled in Yahoo!360 and Microsoft's Xbox 360. Perhaps strict virtual architectures will someday soon be pressed into use for multicore chips. These virtualities are of machinery. Is the principle extensible?

Programmers inhabit the next level up. Virtual programmers would be end users who could routinely put together sequences of high-level statements specific to their problems as users. They wouldn't be constrained by a set of buttons and templates designed for marketing reasons rather than user reasons by people living in a different world who didn't properly understand the problem area. These programmers could put basic operations together in their own sequences, subject to their own conditioning.

This virtual programming could be done through a command and scripting/macro interface. Thus another wheel would turn. ■

***Neville Holmes*** *is an honorary research associate at the University of Tasmania's School of Computing. Contact him at neville.holmes@utas.edu.au. Details of citations in this essay, and links to further material, are at www.comp.utas.edu.au/users/nholmes/prfsn.*

SCHOLARSHIP MONEY FOR STUDENT LEADERS

**Lance Stafford Larson Student Scholarship best paper contest**

✶

**Upsilon Pi Epsilon/IEEE Computer Society Award for Academic Excellence**

**Each carries a $500 cash award.**

***Application deadline: 31 October***

IEEE COMPUTER SOCIETY

Investing in Students

**www.computer.org/students/**