DOI:10.1145/3548781

Peter Denning and Roland Ibbett

# The Profession of IT

# The Atlas Milestone

*Celebrating virtual memory, which has made such a difference in how we approach programming, memory management, and secure computing.*


IMAGE COURTESY OF WWW.IEEE-UKANDIRELAND.ORG

Virtual memory is a technology of computer systems architecture that is as old as academic computer science and has affected the careers of many computing professionals. We take this opportunity celebrate it as a milestone of computing, recognized by the recent IEEE Milestone award to the University of Manchester, where it was invented in 1958.

First, some background. The IEEE is the world's largest technical society with over 430,000 members in 160 countries. The IEEE Milestones program was established in 1983 to recognize the achievements of giants who advanced the electrical and electronics profession around the world. Each IEEE Milestone is recognized by a bronze plaque mounted at the location of the achievement. The IEEE website lists 224 milestones awarded since 1977, of which 35 milestones are associated with computing.[a]

In June 2022 two Milestone plaques were dedicated, one for the "Manchester University 'Baby' computer and its Derivatives 1948–1951" and one for the "Atlas Computer and the Invention of Virtual Memory 1957–1962." An image of the latter plaque appears here.

## The Atlas Computer

The Atlas architecture (see Figure 1) incorporated a multitude of what were then novel features: asynchronous pipelined operation, parallel arithmetic, 128 index registers, double address modification by index registers, extracodes (software sequences simulating additional hardware instructions), interrupts, an interleaved main core store, multiprogramming, and, most importantly, a one-level storage system[2] that later became known as virtual memory. Virtual memory required novel software and hardware, leading to the creation of an operating system known as the Atlas Supervisor.[3] The supervisor also included a compiler for Atlas Autocode, a high-level language similar to Algol 60.

Atlas incorporated multiple kinds of store, including main memory (magnetic core), secondary memory (rotating drum), Fixed Store (precursor to today's firmware holding extra instructions), and V-Store (precursor to today's memory-mapped peripherals). These different kinds of storage were needed to optimize the different storage-access tasks the CPU had to do. The Atlas Processor (CPU) consisted of the Accumulator (called A-register) and its associated floating-point arithmetic unit, the index registers (called B-registers), and the Control section.

Originally called one-level storage,[2] the Atlas virtual memory system gave each user the illusion of having a very large main memory by automating the transfer of code and data between a small fast main core store and a large, much slower, magnetic drum. Prior to this, on earlier Manchester machines, programmers spent vast amounts of time augmenting basic algorithms with "overlay sequences"—calls on the secondary memory to transfer

a See https://bit.ly/3RFg35c

pages (standard size data blocks) into the limited main memory. Kilburn believed the one-level storage mechanism would eliminate manual overlays and estimated that programmer productivity would be improved by a factor of up to 3.

The Atlas allowed every program to address up to 1M words via a 20-bit virtual address. However, this created a problem. Kilburn wrote[2] "In a universal high-speed digital computer it is necessary to have a large-capacity fast-access main store. While more efficient operation of the computer can be achieved by making this store all of one type, this step is scarcely practical for the storage capacities now being considered. For example, on Atlas it is possible to address $10^6$ words in the main store. In practice on the first installation at Manchester University a total of $10^5$ words are provided, but though it is just technically feasible to make this in one level it is much more economical to provide a core store (16,000 words) and drum (96,000 words) combination."

There was more to it than just this, however. In previous machines, page transfers took place under direct program control as directed by the programmer. In Atlas the ratio of drum to processor access time would be approximately 2,700:1,[b] so to avoid having the processor idle for long periods during page transfers, multiple programs were co-resident in the core store at locations hidden from users. When one program stopped for a page transfer, the processor was switched to another resident program. Kilburn's solution to these problems was to make user program addresses virtual addresses and to have the computer itself determine the mapping between virtual and real addresses. The system for implementing this concept was a combination of operating system software and hardware known as paging.

It was clear that translating from virtual to real addresses would have to be done in hardware, otherwise there would be a huge time penalty. Also, it would only be feasible to move blocks or pages of information, rather than individual words, between the drum and core stores. So a set of associative (content addressable) registers, the Page Address Registers, was used (see Figure 2). A PAR held the page number of the page loaded in the associated page-frame of memory. The lock-out bit was set for PARs containing pages of suspended jobs. With the chosen page size of 512 words, the 16K words of core store spanned 32 pages, so 32 PARs were needed. The 20-bit virtual address was therefore split into 11 bits of page address and 9 bits of line address. The page address was presented to all the Page Address Registers simultaneously and in most cases one of them would indicate a match with the presented address. The outputs of the PARs were then encoded to form a 5-bit page-frame address which was concatenated with the 9-bit line address to form the real address to be sent to the core store. All this address mapping was done in a small fraction of a memory cycle and was invisible to programmers.

If there was no PAR match, a page-fault interrupt was generated. The page-fault handler of the operating system intervened, found the missing page on the drum, moved it into a blank page-frame of the main store, and updated that frame's PAR with the page number it now contained. When all this was done, the operating

**Figure 1. The ATLAS architecture.**

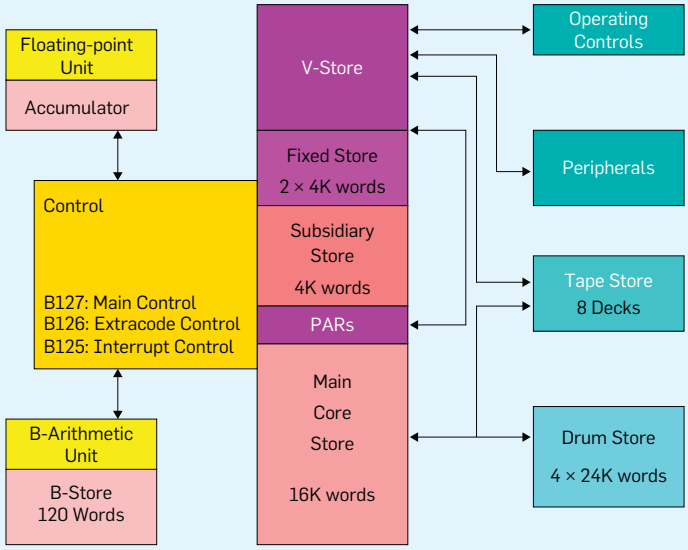
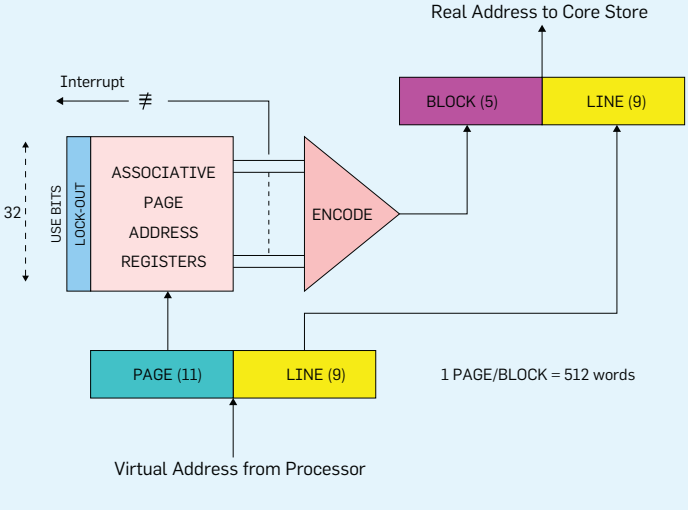



**Figure 2. Page address registers.**




b In today's virtual memories, this ratio is much worse, closer to $10^6$:1

Coming Next Month in COMMUNICATIONS

**Overcoming the Inherent Challenges in Creating a Revolutionary Academic Program**

**Should Young Computer Scientists Stop Collaborating with Their Doctoral Advisors?**

**Linear Address Spaces**

**The Dangers of Participation Bias in Educational Studies**

**Assessing the Quantum Computing Landscape**

**Storytelling and Science**

**AuraRing: Precise Electromagnetic Finger Tracking**

**Traffic Classification in an Increasingly Encrypted Web**

Plus, the latest news about neuro-symbolic logic, finding hidden malware, and how applied AI teaches writing skills.

**The Atlas one-level store was hailed as a major breakthrough in computer systems architecture and was quickly taken up by engineers building other systems.**

system later resumed the interrupted program, which could now continue because its last memory access would now map to main memory. In later virtual memories, not enough PARs could be provided to cover the whole of main memory; the PAR array was replaced with a translation lookaside buffer and a page table.

Now there is one other problem to deal with: maintaining a blank page-frame in memory so that the next page fault had a frame available to receive the missing page. This was done by a replacement policy called the "learning algorithm." As part of processing a page fault, the operating system would use the learning algorithm to select one of the other 31 pages for replacement and initiate a swap to copy that page back to the drum. The learning algorithm was the world's first replacement policy.

The learning algorithm assumed all pages of a program were involved in loops. By monitoring use bits, it measured intervals of use and non-use and calculated a period for each page's loop. It then selected for replacement the page that would not be reused for the longest time into the future. This principle, known today as the "MIN principle," is optimal if indeed all pages are in fixed loops. This assumption is not always met and caused performance problems in virtual memories built after 1962. We will discuss this next.

### Performance of Virtual Memory

The Atlas one-level store was hailed as a major breakthrough in computer systems architecture and was quickly taken up by engineers building other systems. These systems soon encountered two significant performance problems. One was that the best replacement algorithms tended to require heavy overhead—the Atlas "learning algorithm" was of this kind—and the low overhead ones caused too much paging. The other problem was thrashing, an unexpected collapse of throughout in a multiprogrammed system when the number of loaded jobs exceeded an unpredictable threshold. These issues put the entire project for virtual memory under a cloud. What good was a multimillion-dollar computer that was bogged down with paging and whose performance is likely to collapse unpredictably?

In a classic study (1966), Les Belady of IBM put a large number of possible replacement algorithms to the test under a variety of workloads. He concluded the near-zero-overhead FIFO (first in first out) policy generated more paging than most of the others, and that the high-overhead LRU (least recently used) generally outperformed most of the others. He also tested an optimal algorithm, MIN, which gave the least possible amount of paging but was not real-time implementable because it required knowledge of the future. He was disappointed that most of the policies including LRU were significantly poorer performers than MIN. There seemed to be no hope that a paging policy with near optimal performance was feasible.

When these basic algorithms were extended to multiprogramming, the operating system needed to assign a memory region to each job—for example, $N$ jobs would each get $1/N$ of the memory. If $N$ got too high, all the jobs would be pushed into a state of high paging, which meant every job was unable to use the CPU very much and overall throughput collapsed as the jobs "paged to death." There was no way to determine where the threshold $N$ was because it depended on the details of each job.

A breakthrough came in 1966 with the concept of working set.[1] A working set is the intrinsic memory demand of a program—the set of pages that if resident would generate a very low level of paging. The working set was

measured by observing which pages a job accessed in a backward looking window of the last $T$ memory accesses. The working set policy for multiprogramming gave each job enough pages for its working set. This meant a small portion of memory, called FREE, was unused. At a page fault a working set would increase by one page, taking it from FREE. When a page was no longer in a job's $T$-window, it would be evicted and returned to FREE. In this way, no page fault could steal a page from another working set and thereby interfere with its performance. The scheduler would not load a new job if its working set was bigger than FREE. It was impossible for a working set policy to thrash.

The final piece of the performance puzzle—optimal throughput—was provided by the principle of locality. This principle holds that programs access small subsets of their address spaces over relatively long phases, and the entire execution of a program can be described as a series of phases where in each one a locality set of pages was used continuously. If the operating system could detect locality sets and load them, most jobs would generate almost no paging during phases. Most of the paging was caused by the relatively infrequent phase transitions to new locality sets. It is not difficult to prove that such a policy is near-optimal—no other policy, including those with lookahead, can generate significantly higher throughput.[1]

The working set policy does just this because the working set measurement sees exactly the locality set when its $T$-window is contained in a phase. The same locality sets and phases are observed over a wide range of $T$-values. Programs with locality managed by working sets typically operate within 1%–3% of optimal.

It is now easy to see that virtual memory and working-set management are a perfect team to attain best possible, thrashing-free performance from a virtual memory.

### The Secure Kernel Problem

Some people believe virtual memory has become obsolete because memory has become so cheap we can allocate all the real memory a job needs. There is then no paging and the mechanism becomes superfluous.

**Some people believe virtual memory has become obsolete because memory has become so cheap we can allocate all the real memory a job needs.**

While it may be true that most jobs can be fully loaded into main memory, that hardly spells the demise of virtual memory. There are always jobs that are just too big for the available memory. Virtual memory makes it possible to run such jobs.

Even more important, however, is that the address mapping of virtual memory guarantees complete isolation of jobs. A job can access only the page frames linked to its page table, and no frame can be shared between two jobs. Therefore, no job can access the memory held by another job. This default isolation is the basis for security kernels in operating systems. Even if there is no need for an automatic solution to the overlay problem, virtual address mapping provides the logical partitioning that is the basis for secure computing.

What a legacy for Kilburn's invention. C

**References**

1. Denning, P.J. Working set analytics. *ACM Computing Surveys* (Jan. 2021).
2. Kilburn, T. et al. One-level storage system. *IRE Trans. EC-11* (Apr. 1962).
3. Kilburn, T. et al. *The Atlas Supervisor. AFIPS Proc. European Joint Computer Conference (EJCC)* (Dec. 1961).

**Peter J. Denning** (pjd@nps.edu) is Distinguished Professor of Computer Science at the Naval Postgraduate School in Monterey, CA, is Editor of ACM *Ubiquity*, and is a past president of ACM. His most recent book is *Computational Thinking* (with Matti Tedre, MIT Press, 2019).

**Roland Ibbett** (roland.ibbett@bcs.org.uk) is Emeritus Professor of Computer Science at the University of Edinburgh; he was previously Reader in Computer Science at the University of Manchester, where he was a major contributor to the MU5 project.