2005-07

The Profession of IT The Locality Principle

Denning, Peter J.

Association of Computing Machinery

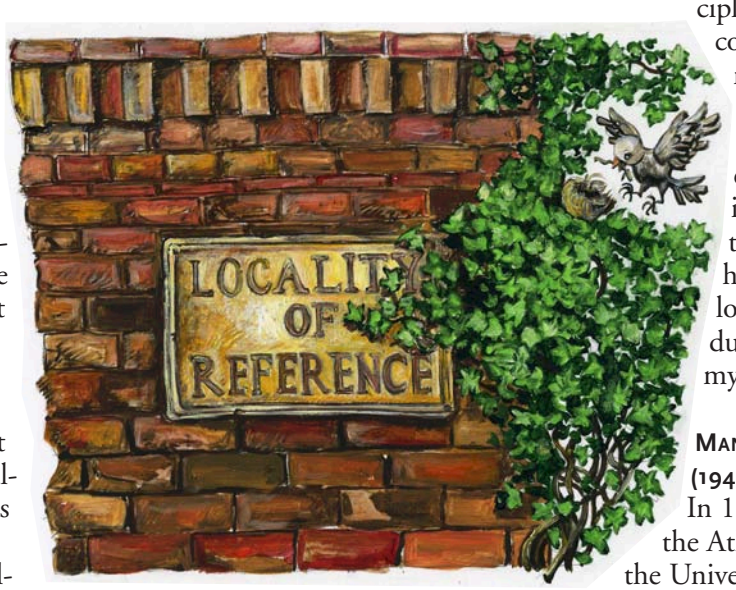
The Locality Principle. (July 2005) Locality of reference is a fundamental principle of computing with many applications. Here is its story.
https://hdl.handle.net/10945/35490

Peter J. Denning

The Locality Principle

Locality of reference is a fundamental principle of computing with many applications. Here is its story.

Locality of reference is one of the cornerstones of computer science. It was born from efforts to make virtual memory systems work well. Virtual memory was first developed in 1959 on the Atlas System at the University of Manchester. Its superior programming environment doubled or tripled programmer productivity. But it was finicky, its performance sensitive to the choice of replacement algorithm and to the ways compilers grouped code onto pages. Worse, when it was coupled with multiprogramming, it was prone to thrashing—the near-complete collapse of system throughput due to heavy paging. The locality principle guided us in designing robust replacement algorithms, compiler code generators, and thrashing-proof systems. It transformed virtual memory from an unpredictable to a robust, self-regulating technology that optimized throughput without user intervention. Virtual memory became such an engineering triumph that it faded into the background of every operating system, where it performs so well at managing memory with multithreading and multitasking that no one notices.



MICHAEL SCHRÖTER

The locality principle found application well beyond virtual memory. Today it directly influences the design of processor caches, disk controller caches, storage hierarchies, network interfaces, database systems, graphics display systems, human-computer interfaces, individual application programs, search engines, Web browsers, edge caches for Web-based environments, and computer forensics. Tomorrow it may help us overcome our problems with brittle, unreliable software.

I will tell the story of this principle, starting with its discovery to solve a multimillion-dollar performance problem, through its evolution as an idea, to its widespread adoption today. My telling is highly personal because locality was my focus during the first part of my career.

Manifestation of a Need (1949–1965)

In 1949, the builders of the Atlas computer system at the University of Manchester recognized that computing systems would always have storage hierarchies consisting of at least main memory (RAM) and secondary memory (disk, drum). To simplify management of these hierarchies, they introduced the page as the unit of storage and transfer. Even with this simplification, programmers spent well over half their time planning and

programming page transfers, then called overlays. In a move to enable programming productivity to at least double, the Atlas system builders therefore decided to automate the overlaying process. Their "one-level storage system" (later called virtual memory) was part of the second-generation Atlas operating system in 1959 [5]. It simulated a large main memory within a small real one. The heart of their innovation was the novel concept that addresses named values, not memory locations. The CPU's addressing hardware translated CPU addresses into memory locations via an updatable page table map. By allowing more addresses than locations, their scheme enabled programmers to put all their instructions and data into a single address space. The file containing the address space was on the disk; the operating system copied pages on demand (at page faults) from that file to main memory. When main memory was full, the operating system selected a main memory page to be replaced at the next page fault.

The Atlas system designers had to resolve two performance problems, either one of which could sink the system: translating addresses to locations; and replacing loaded pages. They quickly found a workable solution to the translation problem by storing copies of the most recently used page table entries in a small high-speed associative memory, later known as the address cache or the translation lookaside buffer. The replacement problem was a much more difficult conundrum.

Because the disk access time was about 10,000 times slower than the CPU instruction cycle, each page fault added a significant delay to a job's completion time. Therefore, minimizing page faults was critical to system performance. Since minimum faults means maximum inter-fault intervals, the ideal page to replace from main memory is the one that will not be used again for the longest time. To accomplish this, the Atlas system contained a "learning algorithm" that hypothesized a loop cycle for each page, measured each page's period, and estimated which page was not needed for the longest time.

The learning algorithm was controversial. It performed well on programs with well-defined loops and poorly on many other programs. The controversy spawned numerous experimental studies well into the 1960s that sought to determine what replacement rules might work best over the widest

1959	Atlas operating system includes first virtual memory; a "learning algorithm" replaces pages referenced farthest in the future [5].
1961	IBM Stretch supercomputer uses spatial locality to prefetch instructions and follow possible branches.
1965	Wilkes introduces slave memory, later known as CPU cache, to hold most recently used pages and significantly accelerate effective CPU speed [9].
1966	Belady at IBM Research publishes comprehensive study of virtual memory replacement algorithms, showing that those with usage bits outperform those without [1].
1966	Denning proposes working set idea: the pages that must be retained in main memory are those referenced during a window of length *T* preceding the current time. In 1967 he postulates that working set memory management will prevent thrashing [2–4].
1968	Denning shows analytically why thrashing precipitates suddenly with any increase above a critical threshold of number of programs in memory. Belady and Denning use term locality for the program behavior property working sets measure [2–4].
1969	Sayre, Brawn, and Gustavson at IBM demonstrate that programs with good locality are easy to design and cause virtual memory systems to perform better than a manually designed paging schedule.
1970	Denning gathers all extant results for virtual memory into *Computing Surveys* paper "Virtual Memory" that was widely used in operating systems courses. This was first coherent scientific framework for designing and analyzing dynamic memories [3].
1970	Mattson, Gecsei, Slutz, and Traiger of IBM publish "stack algorithms," modeling a large class of popular replacement policies including LRU and MIN; they offer surprisingly simple algorithms for calculating their paging functions in virtual memory [7].
1972	Spirn and Denning conclude that phase transition behavior is the most accurate description of locality [8].
1970–74	Abramson, Metcalfe, and Roberts report thrashing in Aloha and Ethernet communication systems; load control protocols prevent it.
1976	Buzen, Courtois, Denning, Gelenbe, and others integrate memory management into queueing network models, demonstrating that thrashing is caused by the paging disk transitioning into the bottleneck with increasing load. System throughput is maximum when the average working set space-time is minimum.
1976	Madison and Batson demonstrate that locality is present in symbolic execution strings of programs, concluding that locality is part of human cognitive processes transmitted to programs [6].
1976	Prieve and Fabry demonstrate VMIN, the optimal variable-space replacement policy; it has identical page fault sequence with working set but lower space-time accumulation at phase transitions.
1978	Denning and Slutz define generalized working sets; objects are local when their memory retention cost is less than their recovery costs. The GWS models the stack algorithms, space-time variations of working sets, and all variable-space optimal replacement algorithms.
1980	Denning gathers the results of over 200 virtual-memory researchers and concludes that working set memory management with a single system-wide window size is as close to optimal as can practically be realized [4].

Table 1. Milestones in development of locality idea.

possible range of programs. Their results were often contradictory. Eventually it became apparent that the volatility resulted from variations in compiling methods: the way in which a compiler grouped code blocks onto pages strongly affected the program's performance under a given replacement strategy.

Meanwhile, in the early 1960s, the major computer makers were drawn to multiprogrammed virtual memory because of its superior programming environment. RCA, General Electric, Burroughs, and Univac all included virtual memory in their operating systems. Because a bad replacement algorithm could cost a million dollars of lost machine time over the life of a system, they all paid a great deal of attention to replacement algorithms.

Nonetheless, by 1966 these companies were reporting their systems were susceptible to a new, unexplained, catastrophic problem they called thrashing. Thrashing seemed to have nothing to do with the choice of replacement policy. It manifested as a sudden collapse of throughput as the multiprogramming level rose. A thrashing system spent most of its time resolving page faults and little running the CPU. Thrashing was far more damaging than a poor replacement algorithm. It scared the daylights out of the computer makers.

The more conservative IBM did not include virtual memory in its 360 operating system in 1964. Instead, it sponsored at its Watson laboratory one of the most comprehensive experimental systems projects of all time. Led by Bob Nelson, Les Belady, and David Sayre, the project team built the first virtual-machine operating system and used it to study the performance of virtual memory. (The term "virtual memory" appears to have come from this project.) By 1966 they had tested every replacement policy that anyone had ever proposed and a few more they invented. Many of their tests involved the "use bits" built into page tables. By periodically scanning and resetting the bits, the replacement algorithm distinguishes recently referenced pages from others. Belady concluded that policies favoring recently used pages performed better than other policies; LRU (least recently used) replacement was consistently the best performer among those tested [1].

1961	IBM Stretch computer uses spatial locality for instruction lookahead.
1964	Major computer manufacturers (Burroughs, General Electric, RCA, Univac but not IBM) introduce virtual memory with their "third-generation computing systems." Thrashing is a significant performance problem.
1965-1969	Nelson, Sayre, and Belady, at IBM Research built first virtual machine operating system; they experiment with virtual machines, contribute significant insights into performance of virtual memory, mitigate thrashing through load control, and lay groundwork for later IBM virtual machine architectures.
1968	IBM introduces cache memory in 360 series. Multics adopts "clock" replacement algorithm, a variant of LRU, to protect recently used pages.
1969–1972	perating systems researchers demonstrate experimentally that the working set policy works as claimed. They show how to group code segments on pages to maximize spatial locality and thus temporal locality during execution.
1972	IBM introduces virtual machines and virtual memory into 370 series. Bayer formally introduces B-tree for organizing large files on disks to minimize access time by improving spatial locality. Parnas introduces information hiding, a way of localizing access to variables within modules.
1978	First BSD Unix includes virtual memory with load controls inspired by working set principle; propagates into Sun OS (1984), Mach (1985), and Mac OS X (1999).
1974–79	IBM System R, an experimental relational database system, uses LRU managed record caches and B-trees.
1981	IBM introduces disk controllers containing caches so that database systems can get records without a disk access; controllers use LRU but do not cache records involved in sequential file accesses.
early 1980s	Chip makers start providing data caches in addition to instruction caches, to speed up access to data and reduce contention at memory interface.
late 1980s	Application developers add "most recent files" list to desktop applications, allowing users to more quickly resume interrupted tasks.
1987–1990	Microsoft and IBM develop OS/2 operating systems for PCs, with full multitasking and working set managed virtual memory. Microsoft splits from IBM, transforms OS/2 into Windows NT.
Early 1990s	Computer forensics starts to emerge as a field; it uses locality and signal processing to recover the most recently deleted files; and it uses multiple system and network caches to reconstruction actions of users.
1990–1998	Beginning with Archie, then Gopher, Lykos, Altavista, and finally Google, search engines compile caches that enable finding relevant documents from anywhere on the Internet very quickly.
1993	Mosaic (later Netscape) browser uses a cache to store recently accessed Web pages for quick retrieval by the browser.
1998	Akamai and other companies provide local Web caches ("edge servers") to speed up Internet access and reduce traffic at sources.

Table 2. Milestones in adoption of locality.

Discovery and Propagation of Locality Idea (1966–1980)

In 1965, I entered my Ph.D. studies at MIT in Project MAC, which was

just undertaking the development of Multics. I was fascinated by the problems of dynamically allocating scarce CPU and memory resources among the many processes that would populate future time-sharing systems.

I set myself a goal to solve the thrashing problem and define an efficient way to manage memory with variable partitions. Solutions to these problems would be worth millions of dollars in recovered uptime of virtual memory operating systems. Little did I know that I would have to devise and validate a theory of program behavior to accomplish this.

I learned about the controversies over the viability of virtual memory and was baffled by the contradictory conclusions among the experimental studies. I noticed all these studies examined individual programs assigned to a fixed memory partition managed by a replacement algorithm. They did not measure the dynamic partitions used in multiprogrammed virtual memory systems. There was no notion of a dynamic, intrinsic memory demand that would tell us which pages of the program were essential and which were replaceable—something simple like, "this process needs p pages at time t." Such a notion was incompatible with the fixed-space policies everyone was studying. I began to speak of a process's intrinsic memory demand as its "working set." The idea was that paging would be acceptable if the system could guarantee that the working set was loaded. I combed the experimental studies looking for clues on how to measure a program's working set.

Figure 1. Locality-sequence behavior diagrammed by programmer during overlay planning.

In an "Aha!" moment during the waning days of 1966, inspired by Belady's observations, I hit on the idea of defining a process's working set as the set of pages used during a fixed-length sampling window in the immediate past. A working set could be measured by periodically reading and resetting the use bits in a page table. The window had to be in the virtual time of the process—time as measured by the number of memory references made—so that the measurement would not be distorted by interruptions. This led to the now-familiar notation: $W(t,T)$ is the set of pages referenced in the virtual time interval of length T preceding time t [2].

By spring 1967, I had an explanation for thrashing. Thrashing was the collapse of system throughput triggered by making the multiprogramming level too high. It was counterintuitive because we were used to systems that would saturate under heavy load, not shut down. I showed that, when memory was filled with working sets, any further increment in the multiprogramming level would simultaneously push all loaded programs into a regime of working set insufficiency. Programs whose working sets were not loaded paged excessively and could not use the CPU efficiently. I proposed a feedback control mechanism that would limit the multiprogramming level by refusing to activate any program whose working set would not fit within the free space of main memory. When memory was full, the operating system would defer programs requesting activation into a holding queue. Thrashing would be impossible with a working set policy.

The working set idea was based on an implicit assumption that the pages seen in the backward window were highly likely to be used again in the immediate future. Was this assumption justified? In discussions with Jack Dennis

(MIT) and Les Belady (IBM), I started using the term "locality" for the observed tendency of programs to cluster references to small subsets of their pages for extended intervals. We could represent a program's memory demand as a sequence of locality sets and their holding times:

(L1,T1), (L2,T2), ... , (Li,Ti), ...

This seemed natural because we knew that programmers planned overlays using diagrams that showed subsets and their time phases (see Figure 1). But what was surprisingly interesting was that programs showed the locality behavior even when it was not explicitly preplanned. When measuring actual page use, we repeatedly observed many long phases with relatively small locality sets (see Figure 2). Each program had its own distinctive pattern, like a voiceprint.

We saw two reasons that this would happen: temporal clustering due to looping and executing within modules with private data; and spatial clustering due to related values being grouped into arrays, sequences, modules, and other data structures. Both these reasons seemed related to the human practice of "divide and conquer"—breaking a large problem into parts and working separately on each. The locality bit maps captured someone's problem-solving method in action. These underlying phenomena gave us confidence to claim that programs have natural sequences of locality sets. The working set sequence is a measurable approximation of a program's intrinsic locality sequence.

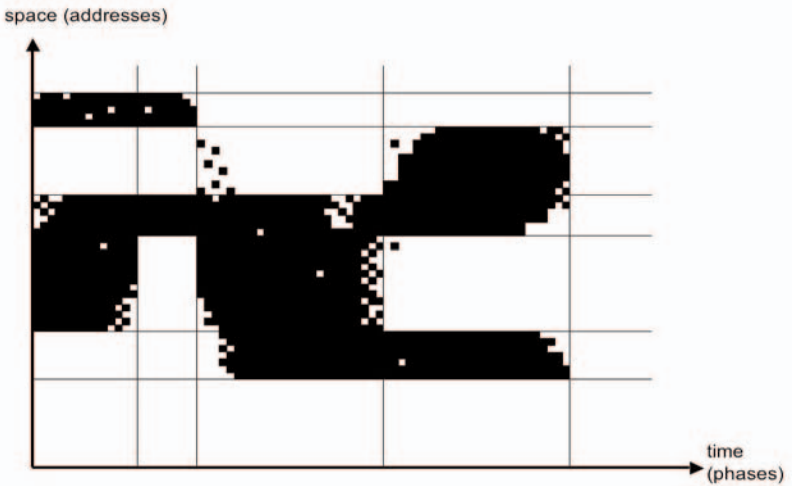


Figure 2. Locality-sequence behavior observed by sampling use bits during program execution.

During the 1970s, I continued to refine the locality idea and develop it into a behavioral theory of computational processes interacting with storage systems within a network of servers. (Table 1 lists key milestones.) By 1980, we defined locality much the same as it is defined today [4], in terms of a distance from a processor to an object x at time t, denoted $D(x,t)$. Object x is in the locality set at time t if the distance is less than a threshold: $D(x,t) \leq T$. Distance can take on several meanings: (1) Temporal: A time distance, such as the time since last reference, time until next reference, or even an access time within the storage system or network. (2) Spatial: A space distance, such as the number of hops in a network or number of addresses away in a sequence. (3) Cost: Any nondecreasing function of time since prior reference.

Adoption of Locality Principle (1967—present)

The locality principle was adopted as an idea almost immediately by operating systems, database, and hardware architects. It was applied in ever-widening circles:

- In virtual memory to organize caches for address translation and to design the replacement algorithms.
- In data caches for CPUs, originally as mainframes and now as microchips.
- In buffers between main memory and secondary memory devices.
- In buffers between computers and networks.
- In video boards to accelerate graphics displays.

- In modules that implement the information-hiding principle.
- In accounting and event logs that monitor activities within a system.
- In the "most recently used" object lists of applications.
- In Web browsers to hold recent Web pages.
- In search engines to find the most relevant responses to queries.
- In spread spectrum video streaming that bypasses network congestion and reduces the apparent distance to the video server.
- In edge servers to hold recent Web pages accessed by anyone in an organization or geographical region.

Table 2 lists milestones in the adoption of locality. The locality principle is today considered a fundamental principle for systems design.

Future Uses of Locality Principle

The locality principle flows from human cognitive and coordinative behavior. The mind focuses on a small part of the sensory field and can work most quickly on the objects of its attention. People gather the most useful objects close around them to minimize the time and work of using them. These behaviors are transferred into the computational systems we design.

The locality principle will be useful wherever there is an advantage in reducing the apparent distance from a process to the objects it accesses. Objects in the process's locality set are kept in a local cache with fast access time. The cache can be a very high-speed chip attached to a processor, a main memory buffer for disk or network transfers, a directory for recent Web pages, or a server for recent Internet searches. The performance acceleration of a cache generally justifies the modest investment in the cache storage.

Two new uses of locality are worth special mention. First, the burgeoning field of computer forensics owes much of its success to the ubiquity of caches. They are literally everywhere in operating systems and applications. By recovering these caches, forensics experts can reconstruct an amazing amount of evidence about a criminal's motives and intent. Criminals who erase data files are still not safe because experts use signal-processing methods to recover the faint magnetic traces of the most recent files from the disk. Not even a systems expert can find all the caches and erase them.

Second, a growing number of software designers are realizing software can be made more robust and less brittle if it can be aware of context. Some have characterized current software as "autistic," meaning that it is uncommunicative, unsocial, and unforgiving. An era of "post-autistic software" can arise by exploiting the locality principle to infer context and intent by watching sequences of references to objects—for example, files, documents, folders, and hyperlinks. The reference sequences of objects, interpreted relative to semantic webs in which they are embedded, can provide numerous clues to what the user is doing and allow the system to adapt dynamically and avoid unwelcome responses.

Early virtual memory systems suffered from thrashing, a form of software autism. Virtual memories harnessed locality to measure software's dynamic memory demand and adapt the memory allocation to avoid thrashing. Future software may harness locality to help it discern the user's context and avoid autistic behavior. C

References

1. Belady, L.A. A study of replacement algorithms for virtual storage computers. *IBM Systems J. 5*, 2 (1966), 78–101.
2. Denning, P.J. The working set model for program behavior. *Commun. ACM 11*, 5 (May 1968), 323–333.
3. Denning, P.J. Virtual memory. *ACM Computing Surveys 2*, 3 (Sept. 1970), 153–189.
4. Denning, P.J. Working sets past and present. *IEEE Trans. on Software Engineering SE-6*, 1 (Jan. 1980), 64–84.
5. Kilburn, T., Edwards, D.B.G., Lanigan, M.J., Sumner, F.H. One-level storage system. *IRE Transactions EC-11* (Apr. 1962), 223–235.
6. Madison, A.W., and Batson, A. Characteristics of program localities. *Commun. ACM 19*, 5 (May 1976), 285–294.
7. Mattson, R.L., Gecsei, J., Slutz, D.R., and Traiger, I.L. Evaluation techniques for storage hierarchies. *IBM Systems J. 9*, 2 (1970), 78–117.
8. Spirn, J. *Program Behavior: Models and Measurements*. Elsevier Computer Science (1977).
9. Wilkes, M.V. Slave memories and dynamic storage allocation. *IEEE Transactions on Computers EC-14* (Apr. 1965), 270–271.

Peter J. Denning (pjd@nps.edu) is the director of the Cebrowski Institute for information and innovation and superiority at the Naval Postgraduate School in Monterey, CA, and is a past president of ACM.