Peter J. Denning. The Locality Principle. In *Communication Networks and Computer Systems* (J. Barria, Ed.). Imperial College Press (2006), 43-67.

CHAPTER 4

The Locality Principle

Peter J. Denning

Naval Postgraduate School
Monterey, CA 93943 USA
Email: pjd@nps.edu

Locality is among the oldest systems principles in computer science. It was discovered in 1967 during efforts to make early virtual memory systems work well. It is a package of three ideas: (1) computational processes pass through a sequence of locality sets and reference only within them, (2) the locality sets can be inferred by applying a distance function to a program's address trace observed during a backward window, and (3) memory management is optimal when it guarantees each program that its locality sets will be present in high-speed memory. Working set memory management was the first exploitation of this principle; it prevented thrashing while maintaining near optimal system throughput, and eventually it enabled virtual memory systems to be reliable, dependable, and transparent. Many researchers and system designers rallied around the effort to understand locality and achieve this outcome. The principle expanded well beyond virtual memory systems. Today it addresses computations that adapt to the neighborhoods in which users are situated, ways to infer those neighborhoods by observing user actions, and optimizing performance for users by being aware of their neighborhoods. It has influenced the design of caches of all sorts, Internet edge servers, spam blocking, search engines, e-commerce systems, email systems, forensics, and context-aware software. It remains a rich source of inspirations for contemporary research in architecture, caching, Bayesian inference, forensics, web-based business processes, context-aware software, and network science.

1. Introduction

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 on to 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 technology that regulated itself dynamically and 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.

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, unforgiving, unreliable, and unfriendly 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, and the attending success of virtual memory, was my focus during the first part of my career.

2. Manifestation of a Need (1949-1965)

In 1949 the builders of the Atlas computer system at 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 [Kilburn]. 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 (Figure 1). 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

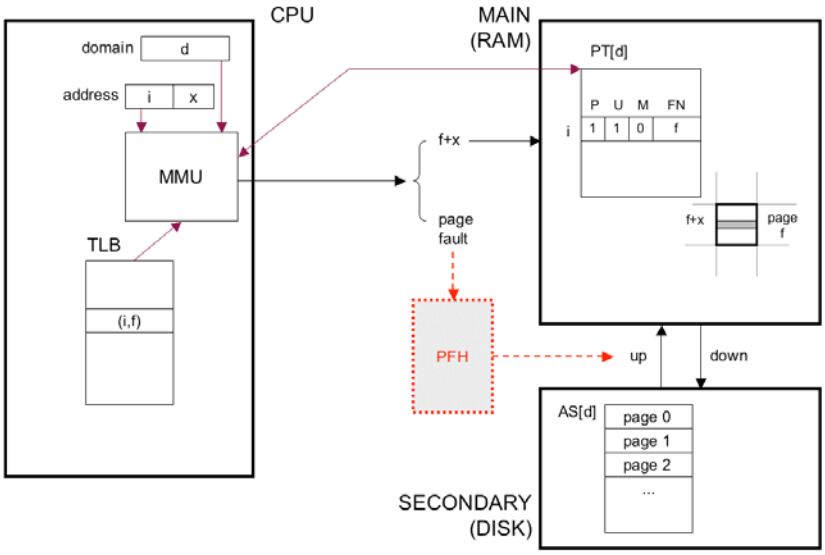

Figure 1. The architecture of virtual memory. The process running on the CPU has access to an address space identified by a domain number d. A full copy of the address space is stored on disk as the file AS[d]; only a subset is actually loaded into the main memory. The page table PT[d] has an entry for every page of domain d. The entry for a particular page (i) contains a presence bit P indicating whether the page is in main memory or not, a usage bit U indicating whether it has been accessed recently or not, a modified bit M indicating whether it has been written into or not, and a frame number FN telling which main memory page frame contains the page. Every address generated by the CPU is decomposed into a page number part (i) and a line number part (x). The memory mapping unit (MMU) translates that address into a memory location as follows. It accesses memory location d+i, which contains the entry of page i in the page table PT[d]. If the page is present (P=1), it generates the memory location by substituting the frame number (f) for the page number (i). If it is not present (P=0), it instead generates a page fault interrupt that signals the operating system to invoke the page fault handler routine (PFH). The MMU also sets the use bit (U=1) and on write accesses the modified bit (M=1). The PFH selects a main memory page to replace, if modified copies it to the disk in its slot of the address space file AS[d], copies page i from the address space file to the empty frame, updates the page table, and signals the CPU to retry the previous instruction. As it searches for a page to replace, the PFH reads and resets usage bits, looking for unused pages. A copy of the most recent translations (from page to frame) is kept in the translation lookaside buffer (TLB), enabling the MMU to bypass the page table lookup most of the time.

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 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 in to page tables

(see Figure 1). 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 [Belady].

3. Discovery and Propagation of Locality Idea (1966-1980)

In 1965, I entered my PhD 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. All these studies examined individual programs assigned to a fixed memory partition managed by a replacement algorithm. They shed no light on the dynamic partitions used in multiprogrammed virtual memory systems. They offered no notion of a dynamic, intrinsic memory demand that would tell 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. All I could find were data on lifetime curves (mean time between page faults as a function of average memory space allocated to a program).

These data suggested that the mean working set size would be significantly smaller than the full program size (Figure 2).

In an "Aha!" moment in 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

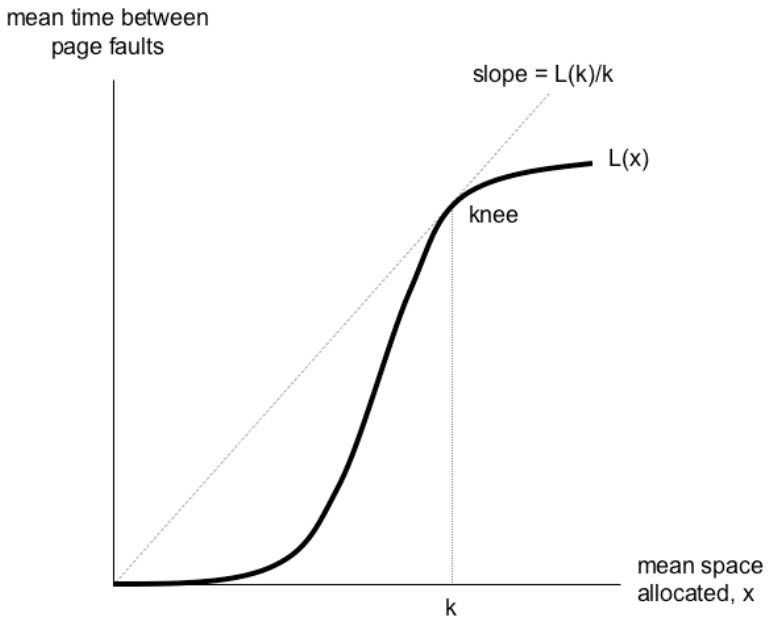

Figure 2. A program's lifetime curve plots the mean time between page faults in a virtual memory system with a given replacement policy, as a function of the amount of space allocated to it by the system. It has an S-shape. The knee, defined as the point at which a line emanating from the origin is tangent to the curve, is the point of diminishing returns for increased memory allocation. The knee memory size is typically considerably smaller than the total program size, indicating that a replacement policy can often do quite well with a relatively small memory allocation. A further significance of the knee is that it maximizes the ratio $L(x)/x$ for all points on the curve. The knee is therefore the most desirable target for space allocation: it maximizes the mean time between faults per unit of space.

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: the working set *W(t,T)* is the set of pages referenced in the virtual time interval of length *T* preceding time *t* [Denning 68a].

By spring 1967, I had an explanation for thrashing [Denning 68b]. 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 (Figure 3). 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, where they paged excessively and could not use the CPU efficiently

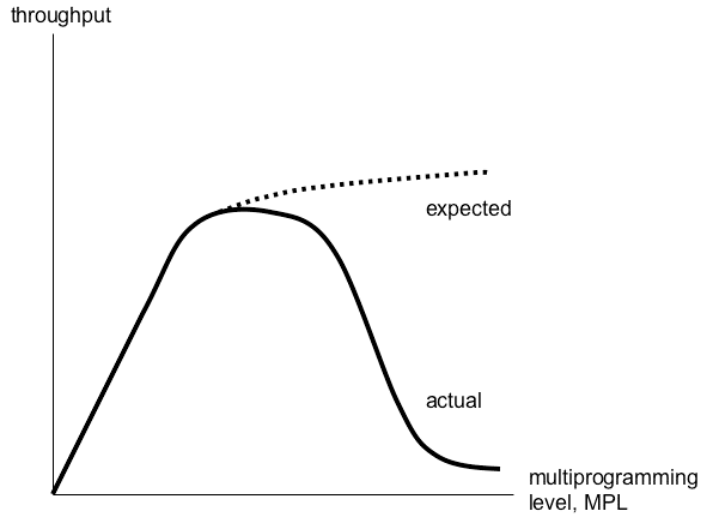

Figure 3. A computer system's throughput (jobs completed per second) increases with multiprogramming level up to a point. Then it decreases rapidly to throughput so low that the system appears to have shut down. Because everyone was used to systems that gradually approach saturation with increasing load, the throughput collapse was unexpected. The thrashing state was "sticky" -- we had to reduce the MPL somewhat below the trigger point to get the system to reset. No one knew how to predict the optimal MPL or to find it without falling into thrashing.

(Figure 4). 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 (Figure 5).

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*,*T2*), (*L2*,*T2*), (*L3*,*T3*), ... , (*Li*,*Ti*), ...

This seemed natural because we knew that programmers planned overlays using diagrams that showed subsets and time phases (Figure 6). But what was strikingly interesting was that programs showed the locality behavior even when it was not explicitly pre-planned. When measuring actual page use, we repeatedly observed many long phases with relatively small locality sets (Figure 7). Each program had its own distinctive pattern, like a voiceprint.

We saw two reasons that this would happen: (1) temporal clustering due to looping and executing within modules with private data, and (2) 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.

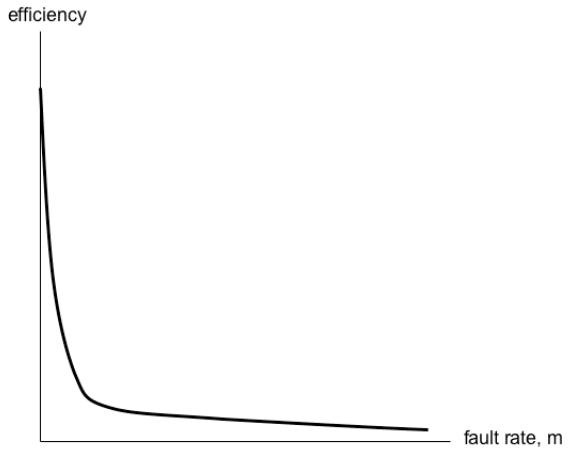

Figure 4. The first rigorous explanation of thrashing argued from efficiency. The efficiency of a program is the ratio of its CPU execution time to its real time. Real time is longer because of page-fault delays. Denote a program's execution time by *E*, the page fault rate by *m*, and the delay for one page fault by *D*; then the efficiency is *E/(E+mED) = 1/(1+mD)*. For typical values of *D* -- 10,000 memory cycle times or longer -- the efficiency drops very rapidly for a small increase of *m* above 0. In a memory filled with working sets (high efficiency), loading one more program can squeeze all the others, pushing everyone into working set insufficiency, collapsing efficiency.

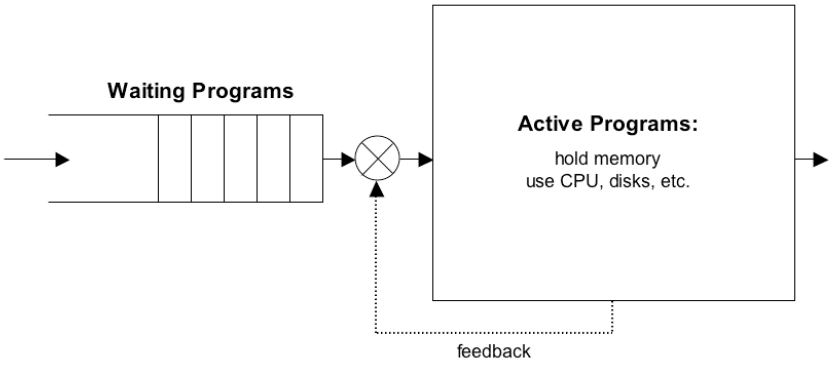

Figure 5. A feedback control system can stabilize the multiprogramming level and prevent thrashing. The amount of free space is monitored and fed back to the scheduler. The scheduler activates the next waiting program whenever the free space is sufficient for its working set. With such a control, we expected that the multiprogramming level would rise to the optimal level and stabilize there.

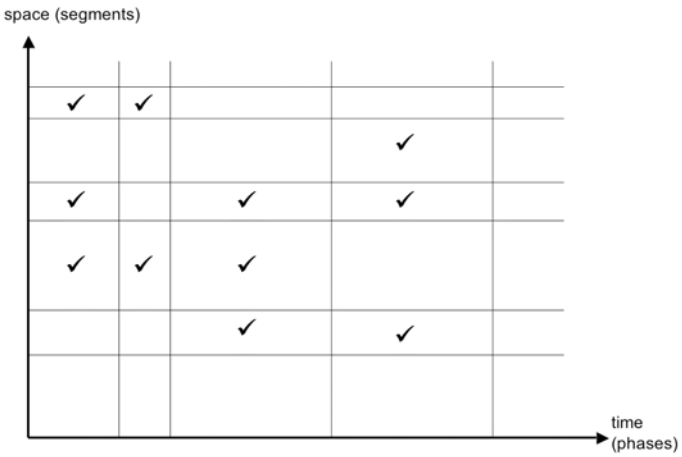

Figure 6. Locality sequence behavior diagrammed by programmer during overlay planning.

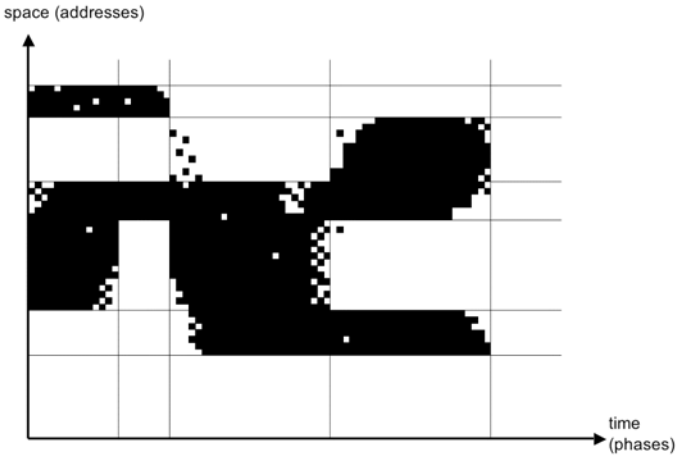

Figure 7. Locality sequence behavior observed by sampling use bits during program execution. Programs exhibit phases and localities naturally, even when overlays are not pre-planned.

As we developed and refined our understanding of locality during the 1970s, I continued to work with many others to refine the locality idea and turn it into a behavioral theory of computational processes interacting with storage systems. By 1980 we articulated the principle as a package of three ideas: (1) computational processes pass through a sequence of locality sets and reference only within them, (2) the locality sets can be inferred by applying a distance function to a program's address trace observed during a backward window, and (3) memory management is optimal when it guarantees each program that its locality sets will be present in high-speed memory [Denning 80]. A distance function $D(x,t)$ measures the distance from a processor to an object x at time t. Distances could be temporal, measuring the time since prior reference or access time within a network; spatial, measuring hops in a network or address separation in a sequence; or cost, measuring any non-decreasing accumulation of cost since prior reference. We said that object x is in the locality set at time t if the distance is less than a threshold: $D(x,t) \leq T$. The storage system would maximize throughput by caching locality sets close to the processor.

By 1975, the queueing network model had become a useful tool for understanding the performance of computing systems, and for predicting throughput, response time, and system capacity. In this model, each computing device of the real system is represented as a server with a queue; the server processes a job for a random service time and then sends it to another server according to a probability distribution for the inter-server transition. The parameters of the model are the mean service times for each server, the mean number of times a job visits a server, and the total number of jobs circulating in the system. We began to use these models to study how to tell when a computing system had achieved its maximum throughput and was on the verge of thrashing. The results were eye-opening.

In the simplest queueing model of a virtual memory system, there is a server representing the CPU and a server representing the paging disk. A job cycles between the CPU and the disk in the pattern

(CPU, Disk) CPU*

meaning a series of CPU-Disk cycles followed by a CPU interval before completing. The number of CPU-Disk cycles is the number of page

faults generated by the system's replacement policy for the mean memory space allocated to jobs. Queueing network theory told us that every server poses a potential bottleneck that imposes an upper limit on the system throughput; the actual bottleneck is the server with the smallest limit. We discovered that the well-known thrashing curve (Figure 3) is actually the system doing the best it can as the paging-disk bottleneck worsens with increasing load (Figure 8.)

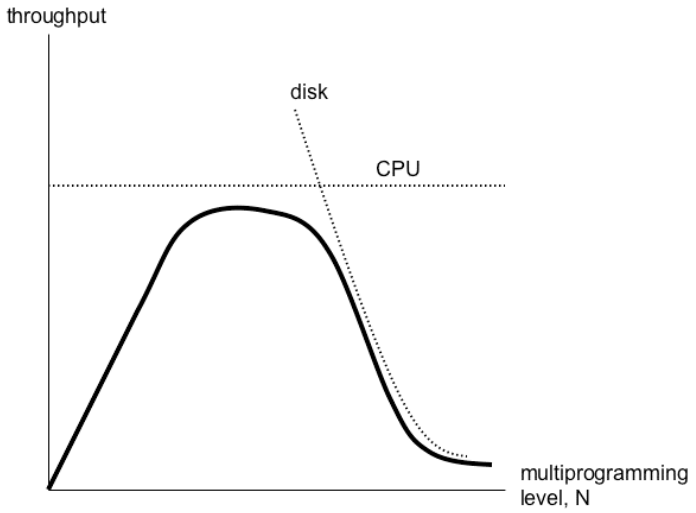

Figure 8. System throughput is constrained by both CPU and disk capacity. The CPU imposes a throughput limit of $1/R$, where R is the average running time of programs. The disk imposes a throughput limit of $1/SF$, where S is the mean time to do a page swap and F is the total number of page faults in a job. Thrashing is caused by precipitous drop of disk capacity as increased load squeezes space and forces more paging. The crossing point occurs when $R=SF$; since $F=R/L$ (lifetime, L), the crossing is at $L=S$, i.e., when the mean time between faults equals the disk service time of a fault. Thus a control criterion is to allow N to increase until L decreases to S. Unfortunately, this was not very precise; we found experimentally that many systems were already in thrashing when $L=S$. Moreover, the memory size at which $L=S$ may bear no relation to the highly desirable lifetime knee (Figure 2).

Once we saw that thrashing is a bottleneck problem, we studied whether we could use bottleneck parameters as criteria for load controls that prevented thrashing. One such criterion was called "$L=S$" because it involved monitoring the mean lifetime L between page faults and

adjusting load to keep that value near the paging disk service time *S* (Figure 8). This criterion was not very reliable: in some experiments, the system would already be thrashing when *L*=*S*. We found that a "knee criterion" -- in which the system adjusted load to keep the observed lifetime near the knee lifetime (Figure 2) -- was consistently more reliable, even though knee lifetime was not close to *S*. Unfortunately, it is not possible to know the knee lifetime without running the program to completion.

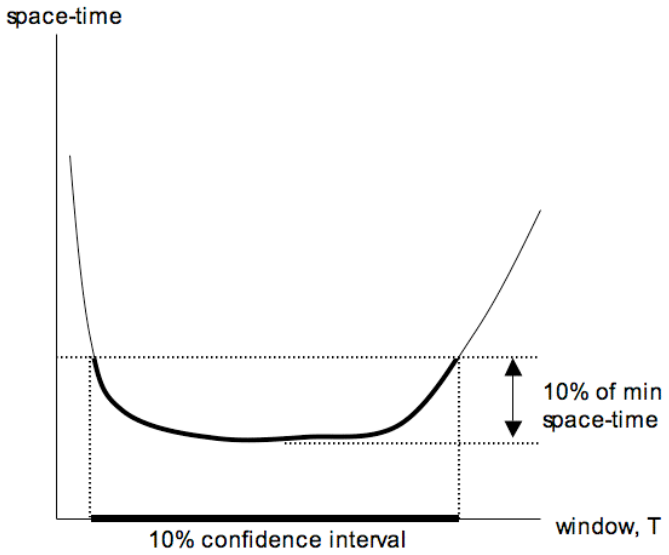

Figure 9. System throughput is maximized when the memory space-time consumed by a job is minimum. The memory allocation that does this is near the knee (Figure 2). Our experimental studies of working-set windows near the knee of its lifetime curve yielded two useful results. One is that a program's space-time is likely to be flat (near minimum) for a broad range of window sizes. The picture shows how we defined a "10% confidence interval" of window sizes.

Our theory told us that system throughput would be maximum when space-time for each job is minimum, confirming our claim that a knee criterion would optimize throughput. How well can a working-set policy approach this ideal? In a line of experimental studies we found that the

interval of window values that put the space-time within 10% of its minimum was quite wide (Figure 9) [Graham]. Then we found that many workloads, consisting of a variety of programs, often had global *T* values that fell in all the 10% confidence intervals (Figure 10). This meant that a single, fixed, properly chosen value of *T* would cause the working set policy to maintain system throughput to within 10% of its optimum. The average deviation was closer to 5%.

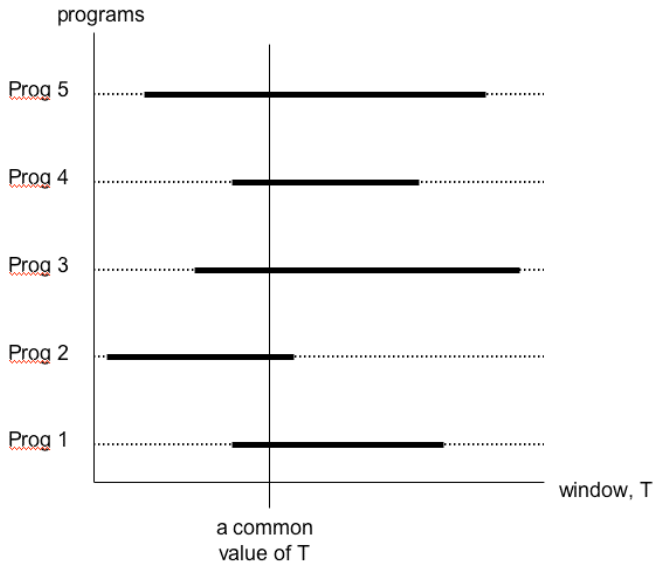

Figure 10. On comparing the 10% confidence intervals, we found that there was very often a global value of *T* that would put all programs within 10% of their space-time minima. The average deviation from minimum for this value of *T* was closer to 5%. The conclusion was that systems with a properly adjusted, single global *T* value would achieve a working-set throughput within 5-10% of optimal.

The final question was: is there another policy that would deliver a lower space-time per job and therefore a higher optimum throughput? Obviously, the VMIN (variable space minimum [Prieve]) would do the job; but it requires lookahead. We discovered that the working set policy has exactly the same page-fault sequence as VMIN. Therefore the difference of space-time between WS and VMIN is completely explained

by working-set "overshooting" in its estimates of locality at the transitions between program phases. Indeed, VMIN unloads pages toward the ends of their phases after it sees they will not be referenced in the next phase. Working set cannot tell this until time *T* after the last reference. Experiments by Alan Smith to clip off these overshoots showed only a minor gain [Smith]. We concluded that it would be unlikely that anyone would find a non-lookahead policy that was noticeably better than working set.

Thus, by 1976, our theory was validated. It demonstrated our original postulate: that working set memory management would prevent thrashing and would allow system throughput to be close to its optimum.

The problem of thrashing, which originally motivated the working set theory, has occurred in other contexts as well as storage management. It can happen in any system where contention for a shared resource can overwhelm the processes' abilities to move forward. It was observed in the first packet-radio communication system, ALOHA, in the late 1960s. In this system, the various contenders could overwhelm the shared spectrum by retransmitting packets when they discovered their transmissions being inadvertently jammed by other transmitters [Abramson]. A similar problem occurred in the Ethernet, where it was solved by the "back-off" protocol that makes a transmitter wait a random time before retrying a transmission [Metcalfe]. A similar problem occurred in database systems with the two-phase commit protocol [Thomasian]. Under this protocol, transactions try to collect locks on the records they will update; but if they find any record already locked, they release all their locks and try again. When too many transactions try to collect locks at the same time, they spend most of their time gathering and releasing locks.

Although it is not critical to the theory and conclusions above, it is worth noting that the working-set analysis applies even when processes share pages. Among its design objectives, Multics supported multiprocess (multithreaded) computations. The notions of locality and working sets had to apply in this environment. The obvious approach was to define a computation's working set as the union of its constituent process working sets. This approach did not work well in the standard paging system architecture (Figure 1) because the use bits that had to be

OR'd together were in different page tables and a high overhead would be needed to locate them. Fortunately, the idea of capability-based addressing, a locality-enhancing architecture articulated by colleagues Dennis and Van Horn in 1966 [Dennis], offered a solution (Figure 11). Working sets could be measured from the use bits of the set of object descriptors.

The two-level mapping inherent in capability addressing is a principle in its own right. It solved a host of sharing problems in virtual memories of multiprocess operating systems [Fabry]. It stimulated a line of extraordinarily fault tolerant commercial systems known as "capability machines" in the 1970s [Wilkes 72, 79]. The architecture was adopted into the run time environments of object oriented programming systems. The principle was applied to solving the problem of sharing objects in the Internet [Kahn]. Thus the situations in which working sets and localities of multithreaded and distributed computations apply are ubiquitous today.

Table 1 summarizes milestones in the development of the locality idea.

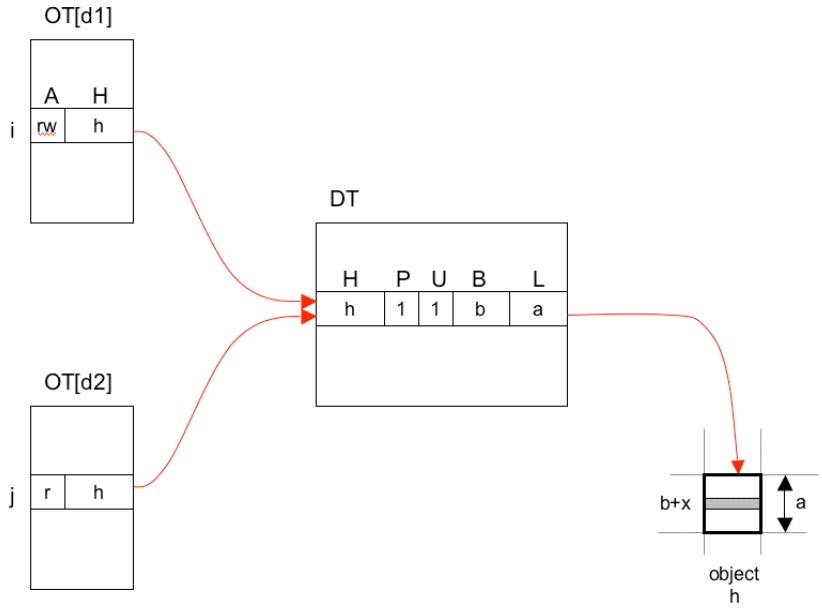

Figure 11. Two-level mapping enables sharing of objects without prior arrangements among the users. It begins with assigning a unique (over all time) identifying handle *h* to an object; objects can be of any size. Object *h* has a single descriptor specifying its status in memory: present (*P* = 0 or 1), used (*U* = 0 or 1), base address (*B*, defined only when *P*=1), and length (*L*). The descriptors of all known objects are stored in a descriptor table DT, a hash table with the handle as a lookup key. When present, the object is assigned a block of contiguous addresses in main memory. Each computational process operates in its own memory domain (such as *d1* or *d2*), which is specified by an object table (OT), an adaptation of the page table (Figure 1). The object table, indexed by an object number (such as *i* or *j*), retrieves an object's access code (such as *rw*) and handle. The memory mapping unit takes an address *(i,x)*, meaning line *x* of object *i*, and retrieves the handle from the object table; then it looks up the descriptor for the handle in the descriptor table; finally it forms the actual memory address *b+x* provided that *x* does not exceed the object's size *a*. Any number of processes can share *h*, simply by adding entries pointing to *h* as convenient in their object tables. Those processes can use any local name (*i* or *j*) they desire. If the system needs to relocate the object in memory, it can do so by updating the descriptor (in the descriptor table). All processes will get the correct mapping information immediately. Working sets can be measured from the use bits (*U*) in the descriptor table.

4. Adoption of Locality Principle (1967-present)

The locality principle was adopted as an idea almost immediately by operating systems, database, and hardware architects. But it did not remain a pure idea for long. It was adopted into practice, 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 in that monitor activities within a system.
- In alias lists that associate longer names or addresses with short nicknames.
- In the “most recently used” object lists of applications.
- In web browsers to hold recent web pages.
- In file systems, to organize indexes (e.g., B-trees) for fastest retrieval of file blocks.
- In database systems, to manage record-flows between levels of memory.
- In search engines to find the most relevant responses to queries.
- In classification systems that cluster related data elements into similarity classes.
- In spam filters, which infer which categories of email are in the user’s locality space and which are not.
- 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.
- In the field of computer forensics to infer criminal motives and intent by correlating event records in many caches.

- In the field of network science by defining hierarchies of self-similar locality structures within complex power-law networks.

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

5. Modern Model of Locality: Context Awareness

As the uses of locality expanded into more areas, our understanding of locality has evolved beyond the original idea of clustering of reference. Today's understanding embraces four key ideas that enable awareness of, and meaningful response to, the context in which software is situated (Figure 12):

- **An observer**;
- **Neighborhoods:** One or more sets of objects that are most relevant to the observer at any given time;
- **Inference:** A method of identifying the most relevant objects by monitoring the observer's actions and interactions and other information about the observer contained in the environment; and
- **Optimal actions:** An expectation that the observer will complete work in the shortest time if neighborhood objects are ready accessible in nearby caches.

The observer is the agent who is trying to accomplish tasks with the help of software, and who places expectations on its function and performance. In most cases, the observer is the user who interacts with software. In some cases, especially when a program is designed to compute a precise, mathematical model, the observer can be built into the software itself.

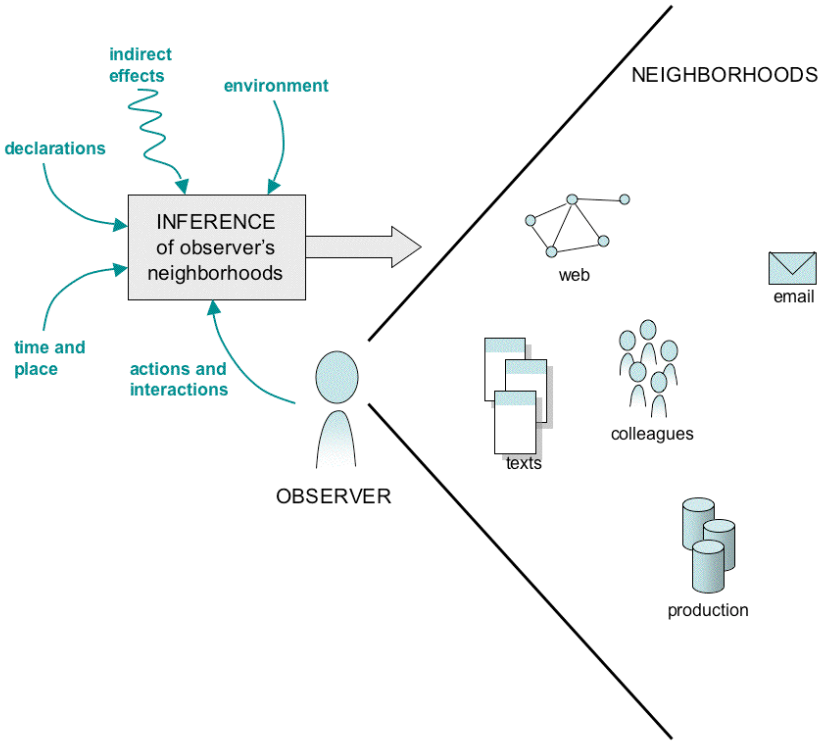

Figure 12. The modern view of locality is a means of inferring the context of an observer using software, so that the software can dynamically adapt its actions to produce optimal behavior for the observer.

A neighborhood is a group of objects standing in a particular relation to an observer, and valued in some way by the observer. The historical example is the locality set associated with each phase of a program's execution; the observer is the process evoked by the program. Modern examples of neighborhoods include email correspondents, non-spam email, colleagues, teammates, objects used in a project, favorite objects, user's web, items of production, texts, and directories.

Some neighborhoods can be known by explicit declarations; for example a user's file directory, address book, or web pages. But most neighborhoods can only be inferred by monitoring the event sequences of an observer's actions and interactions. The event sequences can be

measured either within the software with which the observer is interacting, or outside that software, in the run-time system.

Inference can be any reasonable method that estimates the content of neighborhoods, or at least the most relevant parts. The original working estimated locality sets by observing page reference events in a backward looking window. Modern inference is by a variety of methods such as Google's counting of incoming hyperlinks to a web page, connectionist networks that learn patterns after being presented with many examples, or Bayesian spam filters.

The optimal actions are taken by the system or the software on behalf of the observer. As with the data collection to support neighborhood inference, these actions can come either from inside the software with which the observer is interacting, or from outside that software, in the run-time system.

The matrix below shows four quadrants corresponding to the four combinations of data collection and locus of action just mentioned. Examples of software are named in each quadrant and are summarized below.

		ORIGIN OF DATA FOR INFERENCE	
		Inside	**Outside**
LOCUS OF ADAPTIVE ACTION	**Inside**	Amazon.com, Bayesian spam filter	Semantic web Google
	Outside	Linkers and loaders	Working sets, Ethernet load control

- **Amazon.com; Bayesian spam filters.** This system collects data about user purchasing histories and recommends other purchases that resemble the user's previous purchases, or purchases by other, similar users. Bayesian spam filters gather data about which emails the user considers relevant and then block irrelevant emails. (Data collection inside, optimal actions inside.)

- **Semantic web; Google.** Semantic web is a set of declarations of structural relationships that constitute context of objects and their connections. Programs read and act on it. Google gathers data from the Web and uses it to rank pages that appear to be most relevant to a keyword query posed by user. (Data collection outside, optimal actions inside.)
- **Linkers and Loaders.** These workhorse systems gather library modules mentioned by a source program and link them together into a self-contained executable module. The libraries are neighborhoods of the source program. (Data collection inside, optimal action outside.)
- **Working sets, Ethernet load controls.** Virtual memory systems measure working sets and guarantee programs enough space to contain them, thereby preventing thrashing. Ethernet prevents the contention resolving protocol from getting overloaded by making competing transactions wait longer for retries if load is heavy. (Data collection outside, optimal action outside.)

In summary, the modern principle of locality is that observers operate in one or more neighborhoods that can be inferred from dynamic action sequences and static structural declarations. Systems can optimize the observer's productivity by adapting to the observer's neighborhoods, which they can estimate by distance metrics or other inferences.

6. Future Uses of Locality Principle

Locality principles are certain to remain at the forefront of systems design, analysis, and performance. This is because locality 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 organize their social and intellectual systems into neighborhoods of related objects, and they gather the most useful objects of each neighborhood close around them to minimize the time and work of using them. These behaviors are transferred into the computational systems we design and into the expectations users have about how their systems should interact with them.

Here are seven modern areas offering challenging research problems that locality may be instrumental in solving.

Architecture. Computer architects have heavily exploited the locality principle to boost the performance of chips and systems. Putting cache memory near the CPU, either on board the same chip or on a neighboring chip, has enabled modern CPUs to pass the 1 GHz speed mark. Locality within threaded instruction sequences is being exploited by a new generation of multi-core processor chips. The "system on a chip" concept places neighboring functions on the same chip to significantly decrease delays of communicating between components. Locality is used to compress animated sequences of pictures by detecting the common neighborhood behind a sequence and transmitting it once and then transmitting the differences. Architects will continue to examine locality carefully to find new ways to speed up chips, communications, and systems.

Caching. The locality principle is useful wherever there is an advantage in reducing the apparent distance from a process to the objects it can access. Objects in the process's neighborhood are kept in a local cache with fast access time. The performance acceleration of a cache generally justifies the modest investment in the cache storage. Novel forms of caching have sprung up in the Internet. One prominent example is edge servers that store copies of web objects near their users. Another example is the clustered databases built by search engines (like Google) to instantly retrieve relevant objects from the same neighborhoods as the asker. Similar capabilities are available in MacOS 10.4 (Tiger) and will be in Windows 2006 to speed up finding relevant objects.

Bayesian Inference. A growing number of inference systems exploit Bayes's principle of conditional probability to compute the most likely internal (hidden) states of a system given observable data about the system. Spam filters, for example, use it to infer the email user's mental rules for classifying certain objects as spam. Connectionist networks use it for learning: they internal states that abstract from desired input-output pairs shown to the network; the network gradually acquires a capability for new action. Bayesian inference is an exploitation of locality because it infers a neighborhood given observations of what a user or process is doing.

Forensics. The burgeoning field of computer forensics owes much of its success to the ubiquity of caches. They are literally everywhere in an operating systems and applications. By recovering evidence from these caches, forensics experts can reconstruct (infer) an amazing amount of a criminal's motives and intent [Farmer]. Criminals who erase data files are still not safe because experts use advanced signal-processing methods to recover the faint magnetic traces of the most recent files from the disk [Carrier]. Learning to draw valid inferences from data in a computer's caches, and from correlated data in caches in other computers with which the subject has communicated, is a challenging research problem.

Web Based Business Processes. The principle of locality has pervaded the design of web based business systems, which allow buyers and sellers to engage in transactions using web interfaces to sophisticated database systems. Amazon.com illustrates how a system can infer "book interest neighborhoods" of customers and (successfully) recommend additional sales. Many businesses employ customer relationship management (CRM) systems that infer "customer interest neighborhoods" and allow the company to provide better, more personalized service. Database, network, server, memory, and other caches optimize the performance of these systems [Menasce].

Context Aware Software. A growing number of software designers are coming to believe that most software failures can be traced to the inability of software to be aware of and act on the context in which it operates. The modern locality principle is beginning to enable software designers to reconstruct context and thereby to be consistently more reliable, dependable, usable, safe, and secure.

Network Science. Inspired by A. L. Barabasi, many scientists have begun applying statistical mechanics to large random networks, typically finding that the distribution of node connections is power law with degree -2 to -3 in most cases [Barabasi]. These networks have been found to be self-similar, meaning that if all neighborhoods (nodes within a maximum distance of each other) are collapsed to single nodes, the resulting network has the same power distribution as the original [Song]. The idea that localities are natural in complex systems is not new; in 1976 Madison and Batson reported that program localities have self-similar sub-localities [Madison], and in 1977 P. J. Courtois applied it to

cluster similar states of complex systems to simplify their performance analyses [Courtois]. The locality principle may offer new understandings of the structure of complex networks.

Researchers looking for challenging problems can find many in these areas and can exploit the principle of locality to solve them.

7. References

1. Abramson, N. The ALOHA System--Another Alternative for Computer Communication. Proc. *AFIPS Fall Joint Computer Conference 37* (1970), 281-285.
2. Aho, A. V., P. J. Denning, and J. D. Ullman. Principles of optimal page replacement. *J ACM 18* (Jan 1971), 80-93.
3. Badel, M., E. Gelenbe, J. Lenfant, and D. Potier. Adaptive optimization of a time sharing system's performance. *Proc. IEEE 63* (June 1975), 958-965.
4. Barabasi, A. L. *Linked: The New Science of Networks.* Perseus Books (2002).
5. Belady, L. A. "A study of replacement algorithms for virtual storage computers. *IBM Systems J. 5*, 2 (1966), 78-101.
6. Blake, R. Optimal control of thrashing. *ACM SIGMETRICS Proc. Conf. on Measurement and Modeling of Computer Systems* (1984), 1-10.
7. Brawn, B., and F. G. Gustavson. Program behavior in a paging environment. *Proc. AFIPS Fall Joint Computer Conference 33*. Thompson (1968), 1019-1032.
8. Buzen, J. P. Optimizing the degree of multiprogramming in demand paging systems. *Proc. IEEE COMPCON* (Sep 1971), 139-140.
9. Buzen, J. P. Fundamental laws of computer systems performance. *Acta Informatica 7*, 2 (1976), 167-182.
10. Carr, J., and J. Hennessy. WSCLOCK -- a simple and effective algorithm for virtual memory management. *ACM SIGOPS Proc. 8th Symp. on Operating Systems Principles* (1981), 87-95.
11. Carrier, Brian. *File System Forensic Analysis*. Addison Wesley (2005).
12. Corbato, F. J. A paging experiment with the Multics system. *In Honor of P. M. Morse*, K. U. Ingard, Ed. MIT Press (1969), 217-228.
13. Courtois, P. J. *Decomposability*. Academic Press (1977).
14. Denning, P. J. The working set model for program behavior. *ACM Communications 11*, 5 (May 1968), 323-333.
15. Denning, P. J. Thrashing: Its causes and prevent. *Proc. AFIPS Fall Joint Computer Conference 33*. Thompson (1968), 915-922.
16. Denning, P. J. Virtual memory. *ACM Computing Surveys 2*, 3 (Sept 1970), 153-189.

17. Denning, P. J. Working sets past and present. *IEEE Transactions on Software Engineering SE-6*, 1 (January 1980), 64-84.
18. Denning, P. J. Network Laws. *ACM Communications 47* (Nov 2004), 15-20.
19. Denning, P. J., and S. C. Schwartz. Properties of the working set model. *ACM Communications 15* (March 1972), 191-198.
20. Denning, P. J., K. C. Kahn, J. Laroudier, D. Potier, and R. Suri. Optimal multiprogramming. *Acta Informatica 7*, 2 (1976), 197-216.
21. Denning, P. J., and D. R. Slutz. Generalized working sets for segment reference strings. *ACM Communications 21*, 9 (September 1978), 750-759.
22. Dennis, J. B., and E. C. van Horn. Programming semantics for multiprogrammed computations. ACM *Communications 9*, 3 (March 1966), 143-155.
23. Fabry, R. S. Capability-Based Addressing. ACM *Communications 17*, 7 (July 1974), 403-412.
24. Farmer, D., and W. Venema. *Forensic Discovery.* Addison Wesley (2004).
25. Ferrari, D. Improving locality by critical working sets. *ACM Communications 17* (Nov 1974), 614-620.
26. Gelenbe, E., J. Lenfant, and D. Potier. Analyse d'un algorithme de gestion de memoire centrale et d'un disque de pagination. *Acta Informatica 3* (1974), 321-345.
27. Graham, G. S., and P. Denning. Multiprogramming and program behavior. *Proc ACM SIGMETRICS Conference on Measurement and Evaluation* (1974), 1-8.
28. Hatfield, D., and J. Gerald. Program restructuring for virtual memory. *IBM Syst. J. 10* (1971), 168-192.
29. Kahn, R., and R. Wilensky. A framework for distributed digital object services. Corporation for National Research Initiatives (1995). <http://www.cnri.reston.va.us/k-w.html>
30. Kilburn, T., D. B. G. Edwards, M. J. Lanigan, F. H. Sumner. One-level storage system. *IRE Transactions EC-11* (April 1962), 223-235.
31. Leroudier, J., and D. Potier. Principles of optimality for multiprogramming. *Proc. Int'l Symp. Computer Performance Modeling, Measurement, and Evaluation*, ACM SIGMETRICS and IFIP WG 7.3 (March 1976), 211-218.
32. Madison, A. W., and A. Batson. Characteristics of program localities. *ACM Communications 19*, 5 (May 1976), 285-294.
33. Mattson, R. L., J. Gecsei, D. R. Slutz, I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems J. 9*, 2 (1970), 78-117.
34. Menasce, D., and V. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning.* Prentice-Hall (2000).
35. Metcalfe, R. M., and D. Boggs. Ethernet: Distributed packet switching for local networks. *ACM Communications 19*, 7 (July 1976), 395-404.
36. Morris, J. B. Demand paging through the use of working sets on the MANIAC II. *ACM Communications 15* (Oct 1972), 867-872.
37. Prieve, B., and R. S. Fabry. VMIN -- An optimal variable space page replacement algorithm. *ACM Communications 19* (May 1976), 295-297.
38. Roberts, L. G. ALHOA packet system with and without slots and capture. *ACM SIGCOMM Computer Communication Review 5*, 2 (April 1975), 28-42.

39. Rodriguez-Rosell, J., and J. P. Dupuy. The design, implementation, and evaluation of a working set dispatcher. *ACM Communications 16* (Apr 1973), 247-253.
40. Sayre, D. Is automatic folding of programs efficient enough to displace manual? *ACM Communications 13* (Dec 1969), 656-660.
41. Shore, J. E. The lazy repairman and other models: performance collapse due to overhead in simple, single-server queueing systems. *ACM SIGMETRICS Proc. Intl. Symp. on Performance Measurement, Modeling, and Evaluation* (May 1982), 217-224.
42. Smith, A. J. A modified working set paging algorithm. *IEEE Trans. Computing C-25* (Sep 1976), 907-914.
43. Song, C., S. Havlin, and H. Makse. Self-Similarity of Complex Networks. *Nature 433* (Jan 2005), 392-395.
44. Spirn, J. *Program Behavior: Models and Measurements.* Elsevier Computer Science (1977).
45. Thomasian, A. Two-phase locking performance and its thrashing behavior. *ACM Trans. Database Systems (TODS) 18*, 4 (December 1993), 579-625.
46. Wilkes, M. V. Slave memories and dynamic storage allocation. *IEEE Transactions Computers EC-14* (April 1965), 270-271.
47. Wilkes, M. V. *Time Sharing Computer Systems.* Elsevier (1968, 1972).
48. Wilkes, M. V. The dynamics of paging. *Computer J 16* (Feb 1973), 4-9.
49. Wilkes, M. V., and R. Needham. *The Cambridge CAP Computer and Its Operating System.* North-Holland (1979).

Table 1: Milestones in Development of Locality Idea

1959	Atlas operating system includes first virtual memory; a "learning algorithm" replaces pages referenced farthest in the future [30].
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 [46].
1966	Belady at IBM Research publishes comprehensive study of virtual memory replacement algorithms, showing that those with usage bits outperform those without [5]. Corbato reconfirms for Multics [12].
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 [14,15,17]
1968	Denning shows analytically why thrashing precipitates suddenly with any increase above a critical threshold of number of programs in memory [15]. Belady and Denning use term locality for the program behavior property working sets measure.
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 design paging schedule [7, 40]
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 [16].
1970-71	Mattson, Gecsei, Slutz, and Traiger of IBM publish "stack algorithms", modeling a large class of popular replacement policies including LRU and MIN and offering surprisingly simple algorithms for calculating their paging functions in virtual memory [33]. Aho, Denning, and Ullman prove a principle of optimality for page replacement [2].
1971	Hatfield and Gerald demonstrate compiler code generation methods for preserving locality in executable files [28]. Ferrari shows even greater gains when working sets measure locality [25].
1972	Spirn and Denning conclude that locality sequence (phase-transition) behavior is the most accurate description of locality [44].
1970-74	Abramson, Metcalfe, and Roberts report thrashing in Aloha and Ethernet communication systems; load control protocols prevent it [1,35,38]
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 [3,8,9,13,20,26,31] System throughput is maximum when the average working set space-time is minimum [9,27]

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 [32]. They show that locality sequences have self-similar substructures.
1976	Prieve and Fabry demonstrate VMIN, the optimal variable-space replacement policy [37]; it has identical page faults as working set but lower space-time accumulation at phase transitions [17].
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. [21]
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 [17].
1981	Carr and Hennessy offer effective software implementation of working set by applying sampling windows in CLOCK algorithm [10].
1982-84	Shore reports thrashing in large class of queueing systems [41]. Blake offers optimal controls of thrashing [6].
1993	Thomasian reports thrashing in two-phase locking systems [45].

Table 2: Milestones in Adoption of Locality

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”, an RLU variant, to protect recently used pages.
1969-1972	Operating systems researchers demonstrate experimentally that the working set policy works as advertised. 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 in the Internet very quickly.
1993	Mosaic (later Netscape) browser uses a cache to store recently accessed web pages for quick retrieval by the browser.
1995	Kahn and Wilensky show a method of digital object identifiers based on the locality-enhancing two-level address mapping principle.
1998	Akamai and other companies provide local web caches (“edge servers”) to speed up Internet access and reduce traffic at sources.