Working Set Analytics

PETER J. DENNING, Naval Postgraduate School, Monterey, California

The working set model for program behavior was invented in 1965. It has stood the test of time in virtual memory management for over 50 years. It is considered the ideal for managing memory in operating systems and caches. Its superior performance was based on the principle of locality, which was discovered at the same time; locality is the observed tendency of programs to use distinct subsets of their pages over extended periods of time. This tutorial traces the development of working set theory from its origins to the present day. We will discuss the principle of locality and its experimental verification. We will show why working set memory management resists thrashing and generates near-optimal system throughput. We will present the powerful, linear-time algorithms for computing working set statistics and applying them to the design of memory systems. We will debunk several myths about locality and the performance of memory systems. We will conclude with a discussion of the application of the working set model in parallel systems, modern shared CPU caches, network edge caches, and inventory and logistics management.

CCS Concepts: • **General and reference** → *Cross-computing tools and techniques*; *Performance*; • **Networks** → *Network performance evaluation*; *Network performance modeling*; • **Software and its engineering** → *Software organization and properties*; *Contextual software domains*; *Operating systems*; *Memory management*; *Extra-functional properties*; *Software performance*;

Additional Key Words and Phrases: Working set, working set model, program behavior, virtual memory, cache, paging policy, locality, locality principle, thrashing, multiprogramming, memory management, optimal paging

ACM Reference format:
Peter J. Denning. 2021. Working Set Analytics. *ACM Comput. Surv.* 53, 6, Article 113 (February 2021), 36 pages.
https://doi.org/10.1145/3399709

Virtual memory made its public debut at the University of Manchester in 1962. It was hailed as a breakthrough in automatic memory management and enjoyed an enthusiastic reception. By the mid-1960s, however, operating system engineers had become skeptical of virtual memory. Its performance was untrustworthy and it was prone to unpredictable thrashing. In 1965, I joined Project MAC at MIT, which was developing Multics. The designers of Multics did not want their virtual memory to succumb to these problems. My Ph.D. research project was to find a new approach to managing virtual memory that would make its performance trustworthy, reliable, and near-optimal. The new approach had two parts. The first was the working set model, which was based on the idea of measuring the intrinsic memory demands of individual programs; working sets then determined how many main memory slots were needed and what pages should be loaded into them. The second part was the principle of locality, which was the strong tendency

Author's address: P. J. Denning, Naval Postgraduate School, 1 University Circle, Monterey, CA 93943; email: pjd@nps.edu.

0360-0300/2021/02-ART113 $15.00
https://doi.org/10.1145/3399709

of executing programs to confine their references to limited locality sets over extended phases. After extensive experimental verification, the locality principle was accepted as a universal law of computing. The locality principle made it possible to prove that working-set memory management is near-optimal and immune to thrashing. These discoveries enabled the success of virtual memory on Multics and on commercial operating systems.

Since the 1970s, every operating system textbook has discussed the working set model. Further, every modern operating system uses the working set model as an ideal to underpin its approach to memory management [14]. This can be seen, for example, by looking at the process activity control panels in Windows, where you will see "working set" mentioned in the memory use column. A search of the US patent database shows over 12,000 patents that base their claims on the working set or locality theories.

In this tutorial, I will trace how all this came to be and will show you the powerful analytic algorithms we developed to compute working set statistics and use them to design memory systems.

THE GROWING PAINS OF VIRTUAL MEMORY

In 1959, Tom Kilburn and his team, who were building the Atlas computer and operating system at the University of Manchester, UK, invented the first virtual memory [15]. They put it into operation in 1962 [19]. Virtual memory managed the content of the small main memory by automatically transferring pages between it and the secondary memory. Kilburn argued it would improve programmer productivity by automating the labor-intensive work of planning memory transfers. It was immediately seen by many as an ingenious solution to the complex problems of managing information flows in the memory hierarchy.

The Manchester designers introduced four innovations that soon were adopted as standards in computer architecture and operating systems from then to the present day. One was the *page*, a fixed sized unit of storage and transfer. Programs and data were divided into pages and stored in main memory slots called page frames with copies on the secondary memory. A second innovation was the distinction between *addresses* (names of values) and *locations* (memory slots holding values). The *address space* was a large linear sequence of addresses and the main memory (RAM) a pool of *page frames*, each of which could hold any page from the address space. As the CPU generated addresses into its address space, a hardware *memory mapping unit* translated addresses to locations, using a *page table* that associated pages with page frames. A third innovation was the *page fault*, an interrupt triggered in the operating system when an executing program presented the mapping unit with an address whose page was not in main memory; the operating system located the missing page in secondary memory, chose a loaded page to evict, and transferred the missing page into the vacated page frame. A fourth innovation was the *page replacement policy*, the algorithm that choses which page must be evicted from main memory and returned to secondary memory to make way for an incoming page. The *miss rate function*—fraction of references that produce a page fault—was the key performance measure for virtual memories.

Performance of operating systems has always been a big deal. To be accepted into the operating system, a virtual memory system had to be efficient. The two potential sources of inefficiency were in address mapping and page transfers. They can be called the Addressing Problem and the Replacement Problem.

The Addressing Problem yielded quickly to efficient solutions. A *translation lookaside buffer*, which was a small cache in the memory mapping unit, limited the slowdown from address mapping to 3% of the normal RAM access time. This was an acceptable cost for all the benefits. Virtual memory did indeed double or triple programmer productivity. It also eliminated the annoying problem that hand-crafted transfer schedules had to be redone for each different size of main memory. Even more, virtual memory enabled the partitioning of main memory into disjoint sets, one for each address space, supporting a primal objective to prevent executing jobs from interfering with one another.[1] This was all good news to the designers and users of early operating systems.

The Replacement Problem was much more difficult and controversial. Page transfers were very costly, because each page transfer took as long as the execution of 10^6 or more instructions. It was critical to find replacement policies that minimized page transfers. Early tests brought good news: the page transfer schedules generated automatically by the virtual memory usually generated fewer page moves than hand-crafted schedules [27]. Because the main memory was very expensive and small,[2] even the cleverest programs were likely to generate a lot of page faults. Despite the great pressure on them to find good replacement policies, designers of virtual memories found no clear winners; by 1965 there was considerable heated debate and conflicting data about which replacement policy would be the best. Many engineers began to harbor doubts about whether virtual memory could be counted on to give good performance to everyone using a computer system [26].

The Atlas designers, keenly aware of the high cost of paging, invented an ingenious replacement policy they called the "learning algorithm." It assumed that typical pages spent the majority of their time being visited in well-defined loops. It measured the period of each page's loop and chose for replacement the page not expected to be accessed again for the longest time into the future. The learning algorithm employed the optimality principle that IBM's Les Belady made precise in his MIN paging algorithm in 1966 [2]. Unfortunately, the learning algorithm did not work well for programs that did not have tight, well-defined loops; it did not find favor among the engineers searching for robust paging algorithms.

The innovation of multiprogramming in the mid-1960s added to the consternation about virtual memory performance. With multiprogramming, several jobs could be loaded simultaneously into separate partitions of the main memory, yielding significant improvements of CPU efficiency and system throughput. But multiprogrammed virtual memories brought a host of new problems. How many programs should be loaded? How much memory space should each program get? Should space allocations be allowed to vary? In the process of trying to answer these questions, engineers discovered that virtual memory systems were prone to a new, unexpected, and very serious problem: thrashing.

Thrashing is the sudden collapse of CPU efficiency and system throughput when too many programs are loaded into main memory at once. It is a catastrophic instability triggered when the paging policy steals pages from other programs to satisfy page faults. This condition, originally called "paging to death," could be triggered by adding just one more program to the main memory. The trigger threshold was unpredictable. Who would purchase a million-dollar computer system whose performance could suddenly collapse at random and unpredictable times?

Thus, by 1965, virtual memory appeared doomed. Operating systems designers were facing an enormous challenge. Could they design policies for managing virtual memory that minimized page faults and did not thrash?

The good news is affirmative: the working set policy became a classic ideal for managing memory [14]. All the major modern operating systems—today including Windows, MacOS, and Linux—used memory management policies inspired by the working set model.

[1]The term "job" is used throughout this tutorial to mean any of process, thread, or executing program. It denotes a computational task doing work for a user or the system.

[2]Main memory was very expensive, at least \$0.25 a byte; today a gigabyte (GB) costs \$5.00, about two millionths of that. Even though memory access times have become much smaller, the speed gap between the main memory (RAM) and secondary memory (usually DISK) has risen from 10^4 in 1960 to 10^6 today. Page faults have never been cheap.

INTO THE STORM

I joined MIT Project MAC at the start of the Multics project in 1965. Multics planned to have a multiprogrammed virtual memory. The designers were deeply worried that they could wind up like some of the commercial operating systems that were adopting multiprogrammed virtual memory—hobbled by excessive paging and susceptible to thrashing. I became fascinated with these questions and took them on for my Ph.D. work. Jerry Saltzer posed the research question in a nice way: Considering Multics as a black box, can you design an automatic control mechanism for the virtual memory with a single, tunable, optimizing parameter [13]?

I set out to find answers for the skeptics who seriously doubted that virtual memory could be stabilized. Their primary concern was that no one of a variety of page replacement policies worked consistently well. They also knew that no replacement policy works well for some common problems. For example, matrix multiplication, a mainstay of graphics and neural networks, is very fast if the matrices are all in memory. But if they are stored with rows or columns on separate pages and memory cannot hold them all, then matrix multiplication generates enormous amounts of paging and grinds to a near halt.

In 1966, Les Belady of IBM Watson Research Lab published a famous experimental study comparing a large number of replacement algorithms [2]. One of his findings was that replacement policies relying on use bits to decide which pages to keep in main memory performed better than others.[3] He took this as evidence of a "locality property"—processes tend to reuse pages most recently used in the past. In late 1965 I had independently reached a similar conclusion. I proposed to harness this effect with a "working set", defined as the pages used during a backward-looking sampling window of virtual time.[4] The working set would see the most recently used pages and the operating system would protect them from being paged out [8, 10]. The operating system could prevent thrashing by never allowing a page fault to steal a page from another working set [9]. Belady and I began a collaboration in 1967, in which we postulated that all programs obey a "locality principle" that could be usefully exploited by paging algorithms.

PAGE REFERENCE MAPS

Page reference maps were a useful tool in early virtual memory research. A map shows time on the x-axis and pages on the y-axis. Each point of the time axis represents a sample interval and above it is a column of darkened pixels marking the pages are used in that sample interval. (An example appears in Figure 1.) The sampling interval is T time units, where one time unit is a single page reference. The pages used in a sampling interval are the *locality set* of that interval. A *phase* is a sequence of sampling intervals over which the locality set is unchanged. These maps clearly showed that jobs accessed small subsets of their total address spaces for extended periods. No program with random reference map was ever observed. These maps were strong evidence of a universal locality principle exhibited by all executing programs.

When he undertook his research on paging in Linux around 2009, a half century after the invention of virtual memory, Adrian McMenamin encountered a lot of skepticism among Linux system programmers about locality [24]. They regarded it as an obsolete idea from the early days of virtual memory, no longer relevant because modern computers have so much more memory. To the contrary, McMenamin found out that Linux programs display locality—and it is even more

[3]A use bit is a hardware bit associated with each page frame. When the page is accessed (read or written) the use bit is set to 1 by the addressing hardware. The operating system can scan for used pages and reset the bits to 0. Unused pages are considered inactive and are first to be removed from main memory when space must be freed up.
[4]Virtual time is discrete time measured as number of memory accesses; it is not interrupted by external events such as page faults or other input-output.

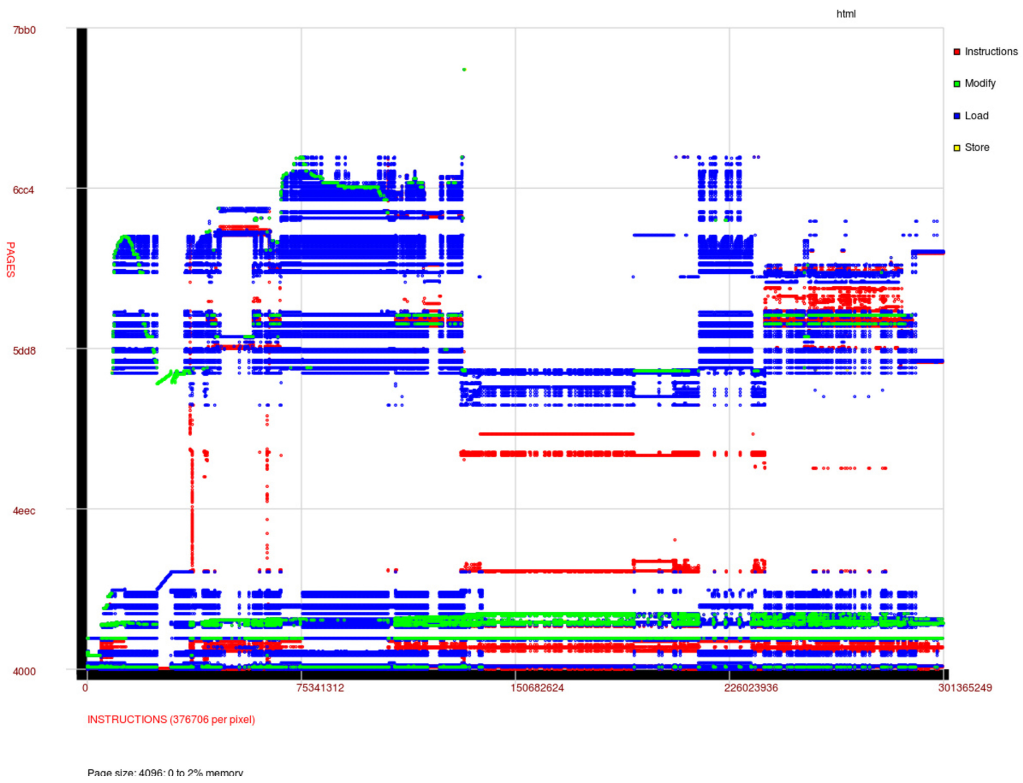


Fig. 1. This is a page reference map of the Firefox Web browser in a Linux system. The horizontal axis represents virtual time, divided into equal sample windows of about 380K references, and the vertical represents virtual addresses of pages. A colored pixel indicates that the page was referenced during the associated sample window; a white pixel indicates no reference. The vertical gridlines are spaced 200 sample intervals apart and the horizontal gridlines are spaced 50 pages apart. The map reveals the locality sets of the program and shows dramatically that locality sets are stable over extended periods (phases), punctuated by sharp shifts to other locality sets. For over 99% of the time in this map, the pages seen in a sample interval are a near perfect predictor for the pages used in the next sample interval. Most executing programs have striking locality maps like this one. Each has its own unique locality behavior, like a digital signature. There is no randomness in the way programs use their code and data. (Source: Adrian McMenamin [24], Creative Commons license.)

pronounced than in early virtual memories. Why would the removal of memory constraints lead to more pronounced locality? The reason appears to be that unconstrained programmers built more modular programs: the phases are intervals of a particular module's use.

McMenamin demonstrated locality by recording the page reference maps from a significant sample of Linux programs. Every program had clearly identifiable locality sets and phases—a unique locality-phase "signature."[5] He concluded that the skepticism was misplaced and that considerable performance benefits will come to systems that exploit the locality behavior. Page reference maps remain a powerful tool for visualizing locality and the operation of memory management policies.

[5]The only known exception is a data scanner, a program that examines each item in a data sequence just once and then discards it. The best replacement policy in this case evicts a page just after it is used.

ORIGINS OF LOCALITY AND WORKING SETS

The ideas of locality and working sets (WS) are separate and distinct but are intimately connected. Locality is about the patterns of programs referencing their pages. Working set is about detecting those patterns in real time and using them to make memory management decisions.

Les Belady's famous study of paging algorithms in 1966 demonstrated that LRU (least recently used) replacement consistently produced fewer page faults than first in first out (FIFO) replacement [2]. Belady reasoned that, if programs localize their references into subsets of their pages, the most recently used pages were most likely to be reused in the immediate future—and thus the least recently used pages were the best choices for replacement. Belady also noted that a weaker form of locality—non uniform use of pages—explained why the fault rate functions of LRU and FIFO were not linear.

By 1966 there was an emerging consensus among operating system engineers that locality was an observed tendency for programs to reuse their pages in the immediate future. I saw a connection between this informal idea of locality and an old programming concept. Programmers used the term "working set" to mean the pages that needed to be loaded in main memory so that a program would execute efficiently. It was up to the programmer to declare the working sets and design a schedule of page transfers to ensure that working sets were loaded in main memory. It seemed to me that the operating system could detect working sets by monitoring use bits, enabling it to automatically load working sets in memory without having to ask programmers to declare them. Pages whose use bits were set during a window of fixed size would estimate the program's working set [8].[6] Thus the two ideas, locality and working set, became a powerful partnership for allocating memory.

Let us examine locality more closely. From page reference maps, early virtual memory researchers saw that programs used only a small subset of their pages at any given time. The pages that were used together were seen as "spatially clustered," because the use of one implied that another would be used soon. For example, the pages of a loop or the pages of a code module are spatially clustered. Spatial clustering implied temporal clustering. These two terms became favorite ways of explaining why locality was a characteristic of program execution.

In the decade after 1968, my students and I, along with other researchers, studied how locality is manifested in actual programs. We studied and measured many programs, leading us to introduce the terms "locality sets," "phases," and "transitions." These terms captured the recurring structure of locality – periods of stability punctuated by abrupt transitions.

We concluded that temporal clustering is a rather poor description of the punctuated stability observed in executing programs. Temporal clustering exemplifies "slow drift" but not "sudden change." To confirm this, we built and studied mathematical models of locality. My student Jeff Spirn tested the "slow drift" hypothesis by formulating a series of mathematical models of slow drift behavior and then examining how well each model predicted the observed LRU miss-rate function of a program [28]. Spirn found that simple models such as independent reference model (each page is referenced with a fixed probability) and independent stack distance model (each LRU stack distance occurs with a fixed probability) led to poor predictions of LRU miss rate.

My student Kevin Kahn modeled the punctuated stability behavior directly [18]. In his models, states were locality sets and phases were holding times in the states. Phase-transition models parameterized from measurements of page reference maps yielded excellent agreement between

[6]Although WS and LRU favor most-recently-used pages, they are not the same. LRU operates within a fixed memory space assigned by the operating system but does not advise the operating system what size it should be. WS measures the locality set and advises the operating system to allocate just the right amount of memory required to hold it. The WS contents are the most recently used pages in the window, but there the similarity with LRU ends.

predicted and actual LRU miss rates. Moreover, because it adjusted to locality sets, a WS policy generated less paging than LRU without using more memory.

Wayne Madison and Alan Batson confirmed that these key aspects of locality—locality sets, phases, and transitions—exist at the source code level [22]. They concluded that locality is not an artifact of the way that compilers lay out data and code blocks on pages of address space. Design techniques such as loop iteration, divide-and-conquer, and modularity lead to subsets of pages being used for extended periods. The locality seen in page references is the image of the higher-level locality created as programmers design their algorithms.

Programmers who understood that virtual memory performs better with programs of good locality easily designed programs that ran well in virtual memory [27]. Every program we ever saw exhibited locality. No program used its pages randomly. By the mid-1970s, we had settled on the definition of locality in the accompanying box [13].

The Locality Principle

Executing processes reference their memory objects with punctuated stability described by:

$$(L_1, H_1),\ (L_2, H_2), \ldots, (L_i, H_i), \ldots$$

where L_i is a locality set and H_i is the holding time of its phase. The shortest sampling interval required to see the full locality set is likely to be a small fraction of holding time. Successive locality sets are likely to be mostly different, with few overlaps.

Recent research by Chen Ding and his students with large shared caches has demonstrated that locality is observed in cache reference patterns. Consequently, working set memory management also applies to shared caches, just as in multiprogrammed operating systems [31, 32, 33].

I have found that students today often have trouble understanding the principle of locality. Many are like McMenamin before his study: it seems counterintuitive that programs have such pronounced locality behavior, or that tracking locality optimizes performance. This misunderstanding may be rooted in their own experience of programming: They were not concerned about memory constraints and did not consciously design their programs to have long phases of stable references to objects. Yet the experimental evidence repeatedly shows that their programs display the phase-transition behavior of locality.

This misunderstanding is supported by the limited definitions of locality in operating systems textbooks. The books usually define locality as a combination of temporal and spatial closeness of references. These definitions ignore the empirical fact of abrupt changes at phase transitions. Notice that random assignment of data to pages might remove spatial locality but it will not remove temporal locality. The same phases and transitions will be observed on the page reference map.

Let us illustrate with Figure 1 how we can measure some properties of locality behavior from a page reference map. The figure shows five locality sets and their associated phases for a sampling window T of approximately 380K time units. A measurement of the graph yields Table 1, where size is the number of pages in the locality set, length is the number of sample intervals in a phase, and fraction is the percentage of the full address space covered by the locality set. Cache policies that detect locality sets in Figure 1 would need only memory sufficient to hold 15–33% of the address space to achieve 100% of the performance.

Notice that the punctuated-stability idea of locality easily translates into measurements of locality set sizes, phase lengths, and transition probabilities on page reference maps, whereas the vague terms temporal and spatial locality do not suggest measurement protocols.

Table 1. Phases of the Firefox Program

set	size	length	fraction
1	50	180	25%
2	65	220	33%
3	30	220	15%
4	55	50	28%
5	35	180	18%

Page reference maps, which were a useful tool in the early studies of virtual memory performance, continue to be useful today in studies of cache performance. They are striking evidence of locality and visually convey considerable insight into the dynamics of executing programs.

Like other scientific discoveries, the locality principle began as a hypothesis and was accepted as scientific fact only after many validations. The original notion of "slow drift" temporal locality gave way to a more sophisticated notion of "punctuated stability" characterized by phases and transitions. Working set is a real-time detector of this behavior. It allows a virtual memory manager to dynamically adjust memory allocations and maximize system throughput.

MEMORY SPACE-TIME LAW

Operating system designers have always been concerned with performance. They would like to state performance guarantees for throughput and response time that will hold over a wide range of workloads and numbers of simultaneous users. One of the most challenging questions was how to establish a connection between memory management and the key performance measure of throughput. When can we say that optimizing memory management optimizes system throughput?

When jobs use less space, we can get more of them into memory and increase throughput because of the parallelism. And when they complete in less time, we can process more of them over time. This is why many of us have an intuition that the smaller the space-time footprint of programs, the larger is the system throughput.

The space time footprint is the number of page-seconds of memory usage of an executing job. A page-second is a unit of rent for using memory. It is analogous to the idea of charging for office space by the number of square-feet rented, or charging labor on a project by person-hours. When a process loads 1 page into main memory for S seconds, the process adds S page-seconds to its memory bill. Loading S pages for 1 second also adds S page-seconds to the bill.

Jeff Buzen in 1976 discovered the "memory space-time law" (MST Law), an exact formula that links space-time and system throughput [5]. It says "average total memory used by all jobs = system throughput × average space-time footprint per job." In symbols,

$$M = XY$$

The proof is simple. Measure the system in an interval of length T. Let Z denote the total space-time used by all the jobs in that interval. (For fixed memory, $Z = MT$.) The throughput is the number of jobs completed in that interval (C) divided by the length of the interval: $X = C/T$. The mean space-time footprint of a job is $Y = Z/C$. The product of X and Y is obviously $M = Z/T$.[7]

[7]The memory space-time law can be seen as an instance of Little's law. Little's law says that the mean number in system is the product of the throughput and the system response time. We can interpret M as the mean number of pages in the system; X as the throughput; and Y as the aggregate holding time accumulated by a job for all its pages.

The MST Law is general and applies to any system or network that has memory usage and throughput. Some people find it hard to believe that the relation between memory usage and throughput is this simple. It really is.

The obvious conclusion is that a policy that minimizes space-time will maximize throughput. To make it easy to tell when this is happening, we will, in the analytics to follow, use space-time to measure the memory usage of working sets and other memory policies. If needed, we can easily convert space-time measures to time averages by dividing space-time by the length of time of the measurement.

There is a complication. The space-time measures of memory policies are defined in virtual time. The space-time needed for the MST Law is defined in real time. We need a way to convert virtual space-time to real space-time.

The most accurate way to do this is with the help of a queueing network model [12]. The model would account for all delays beyond virtual time, such as input-output and page transfers. Setting up such a model is beyond the scope of this tutorial.

However, a good approximation can be made simply by augmenting virtual space-time with the space-time accumulated while servicing page faults. To do this, we require these quantities:

D = page fault delay, typically 10^6 memory accesses
N = length of virtual time a process is executed
S = virtual space-time accumulated by the process during execution
C = number of page faults accumulated during execution
m = miss rate, C/N.

The space-time cost of one page fault is the mean space S/N times the delay D. For all page faults it is $(S/N)(D)(C) = (S)(D)(C/N) = SDm$. Therefore the total real space-time is estimated as $S+SDm$, or in the notation of the MST Law,

$$Y = S(1 + Dm)$$

Thus the real space-time is approximately the virtual space time dilated by the factor $1+Dm$. For example, if the miss rate is10^{-4} (1 fault in 10^4 memory accesses), the dilation factor is 101.

Policies such as LRU or FIFO work with a fixed number k of pages; their total virtual space-time is kN. Thus, Y is smaller only when m is smaller and the policy with lowest miss rate will have highest system throughput. But there is more to the story. Once the miss rate as a function of k is known, there is a value of k that minimizes the expression for Y [13]. Even for the optimal policy, there is a best memory size that minimizes the space-time.

When we discuss the variable-space policy working set, we will see that minimizing virtual space-time maximizes system throughput.

TWO CONTRASTING VIEWS OF MEMORY MANAGEMENT

The familiar way of looking at the memory sees a single CPU (representing a job in execution) accessing pages in a fixed-size memory. No other CPU or memory region is visible. This is called the Fixed Memory View (FMV) (see Figure 2). In this view, the job gets a fixed space in main memory and is unaffected by the presence of other jobs in the memory. The performance of a replacement policy is then completely determined by its total fault count function.

Extensive experimental studies and experience with operating systems led designers to favor two basic replacement policies. FIFO treats the memory space of k pages as a FIFO queue and ignores page usage. Its primary attraction is simplicity and almost negligible overhead. LRU (least recently used) treats the memory as a container of the k most recently used pages; at a page fault, it replaces the page in memory that has not been used for the longest time. Although LRU generates

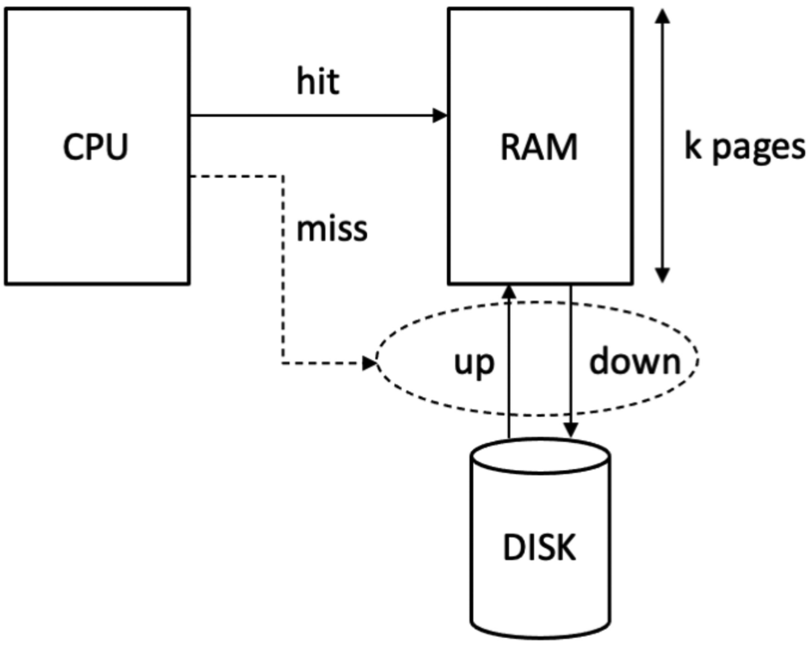


Fig. 2. The FMV sees the system as a single CPU accessing a single RAM (main memory) of fixed size k pages. Address mapping hits are translated directly to RAM addresses. Address mapping misses trigger page faults that cause up and down page moves between RAM and DISK (secondary memory). The objective is to efficiently compute the fault function $F(k)$, which counts the number of page faults when memory size is k pages, and then select replacement policies that minimize $F(k)$. In a system with multiprogramming, the RAM visible in this view is a fixed region of the full RAM.

fewer page faults than FIFO, LRU has a high implementation overhead. Many analysts believed that the performance gains of LRU were cancelled by its overhead.

An elegant compromise between FIFO and LRU is called CLOCK. It treats the FIFO queue as a circular list of size k with a scanning pointer; the page names are analogous to the numerals on rim of a clock and the pointer to the clock's hand. At a page fault, the operating system moves the hand along the list, skipping over those with use bits on (and resetting them). When it finds a page with use bit off, it selects that page for replacement. CLOCK has overhead comparable to FIFO and performance comparable to LRU. CLOCK is commonly used in operating systems. (In early virtual memory systems, CLOCK was called FINUFO, for first-in-not-used-first-out [8].)

In 1970, Richard Mattson and his IBM colleagues discovered a highly efficient way to represent a large class of paging algorithms and compute their fault functions. They called their theory "stack algorithms" [23]. A stack algorithm is a paging policy whose memory contents can be represented with a single list of all the job's pages, called the stack, such that the contents of k-page memory are the first k elements of the stack. LRU's stack lists all the job's pages from most to least recently used; the contents of the k-page LRU memory are the k most recently referenced pages. At each page reference, LRU's stack is updated by moving the referenced page to the top and pushing the intervening pages down one position. Another famous stack algorithm is Belady's optimal MIN algorithm, which replaces the page with longest time until next reference. For the MIN stack algorithm, the update is slightly more complex than LRU: The referenced page moves to the top and the intervening pages are rearranged downward according to their relative priorities assigned by MIN. For any stack algorithm, the position of the next reference in the stack is called stack distance. The fault function $F(k)$ can be computed simply as the number of stack distances larger than k. This elegant theory is frequently discussed in operating systems textbooks.

Before the stack theory was invented Les Belady and his IBM colleagues discovered that FIFO could increase page faults signficantly when more space was allocated [3]. This undesirable behavior is called "Belady's Anomaly." Stack algorithms obey an "inclusion property" and are free of this anomaly. It is unknown whether CLOCK is a stack algorithm.

The stack theory did not answer two important questions: What is the optimal amount of memory to allocate? How is system throughput related to the fault function $F(k)$? These questions were answered by other modeling techniques; details are in Reference [13].

Unfortunately, the FMV and its analytic theory is not very helpful for real operating systems, which allow multiple jobs to reside in RAM at the same time. The FMV gives no insight into important questions including the following:

- How to partition the RAM among the jobs?
- How much space to give each job?
- How to manage variations in the space allocations?
- How to organize page replacement to maximize system throughput?
- How to manage interactions among jobs such as a page fault in one stealing a page from another?
- How to prevent thrashing, a collapse of system throughput when too many jobs are loaded into RAM?

We obviously need a different way of thinking about the common situation of multiprogramming. It is called the shared memory view (SMV) (see Figure 3).

Because of all the variables involved, the optimal management of shared memory cannot be inferred from fixed memory. Obvious extensions of the fixed memory view lead to poor performance and instability. For example, many operating systems simply extended the fixed memory view to include all of RAM. The resulting global LRU would replace the oldest unused page in RAM regardless of which job it belongs to. Unfortunately, the global list of pages does not reflect actual recency of use within jobs. It is ordered mainly by the round-robin scheduler of the ready list: the pages atop the global LRU stack belong to the job that most recently received a time slice. If paging activity is high, then by the time a job cycles back to the front of the ready list some of its pages have been removed. The cascading effect causes high paging in every job, pushing whole system into a state of "paging to death," in which every job spends most of its time queued at the DISK and CPU throughput collapses. This condition is called thrashing (see Figure 4).

We need a different way of thinking about managing shared memory. The working set model provides the basis. We need to abandon the fixed memory view and instead use the working set interpretation of shared memory view to see what is going on. The remainder of this article will approach this in two stages. First, we will define analytic methods for computing working set statistics, such as miss rate, from a given address trace. These methods do not depend on locality or any other assumptions about program behavior. Second, we will show that the working set policy is close to optimal for programs whose address traces conform to the principle of locality. In that case, the working set space-time is very close to the optimal space-time.

WORKING SETS

Originally the term "working set" was an informal term meaning the smallest set of a job's pages that needed to be loaded in main memory for efficient execution. The working set model gave a precise definition in terms of the page-reference behavior of an executing job in a moving sampling

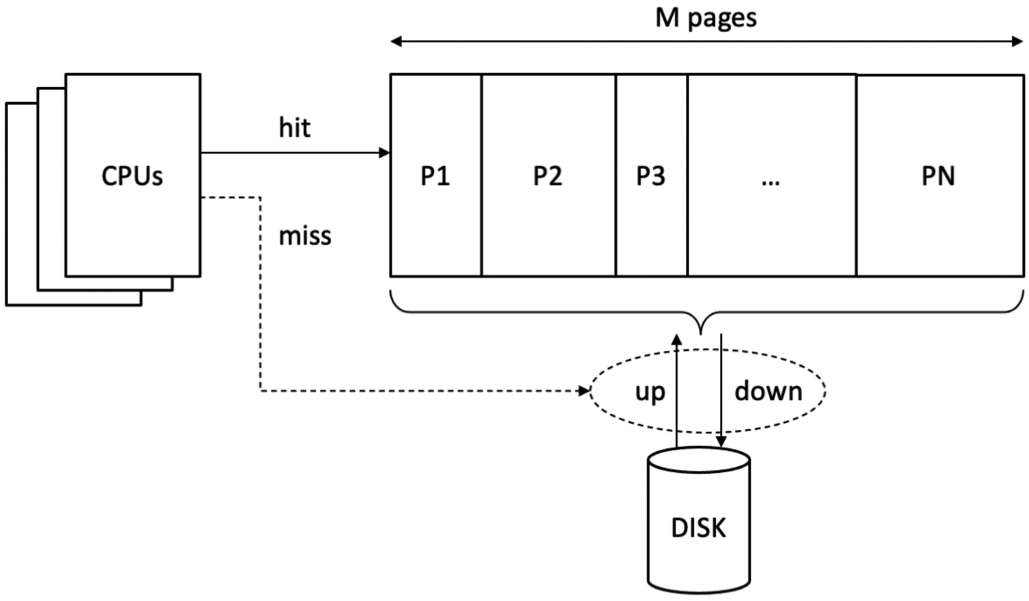


Fig. 3. The SMV sees the system as a set of CPUs (one for each job) sharing the *M*-page RAM. The memory is partitioned among *N* jobs, each getting its own set of page frames disjoint from all the others. A page fault may trigger the page replacement algorithm to steal a page from another job, thereby increasing the space occupied by the faulter and decreasing the space occupied by the victim.

window. The executing job itself informs us of what memory it needs, without regard for external factors such as interrupts. Thus, working sets are a measure of the intrinsic, dynamically varying demand of a computation for memory [8, 10].

The formal definition is that the working set at a particular time t is the pages referenced in a backward-looking window of size T including time t. It is denoted $W(t,T)$. The size, $w(t,T)$, is the number of distinct pages in $W(t,T)$; the size is always at least 1 and never larger than T. Size may be considerably smaller than T due to repeated references to the same pages with the window. Figure 5 illustrates.

The working set window can be thought of as a *lease*. A lease is a guarantee to hold a page in RAM for minimal period of time T. A page's lease can be stored in a timer register associated with a page frame. When a page is loaded into RAM or reused while in RAM, its lease is reset to T. It ticks down to 0 after T time units of non-use. When the lease runs out, the page is evicted from the working set. The lease definition is equivalent to the window definition [20].

THE WORKING SET POLICY

Working set memory management partitions the main memory into dynamically changing regions, one for the working set of each job using the memory. The basic idea is to maintain each job's working set and not allow any other process to steal pages from it [8] (see Figure 6). This is implemented by maintaining a free space in memory, usually smaller than any of the working sets. When a job references a page not in its working set (a page miss or fault), the operating system transfers a page from free space to the job, increasing its working set size by one. In parallel, when a page times out from its job's window T, the operating system transfers it back to the free space, decreasing the working set by one. In this regime page faults and evictions need not coincide as they do when memory allocation is fixed.

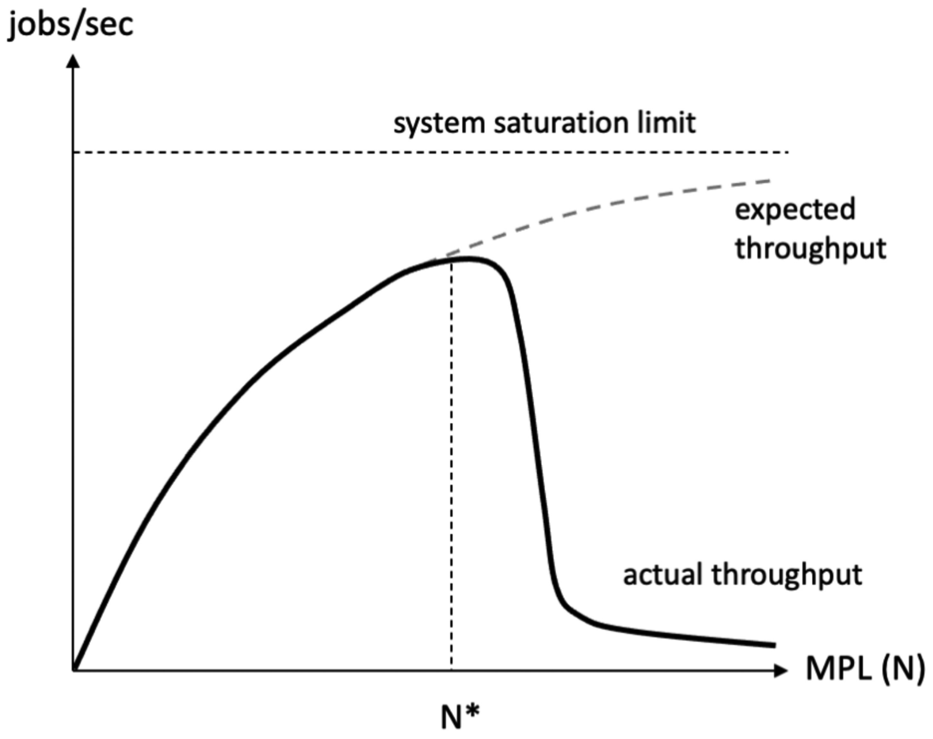


Fig. 4. Thrashing is the collapse of system throughput when too many jobs are loaded into RAM at once. It is a chaotic condition whose onset cannot be predicted accurately—the trigger threshold N^* is unpredictable and very sensitive. Loading one additional job can trigger thrashing. This can happen in any shared memory, such as cache, not just RAM.

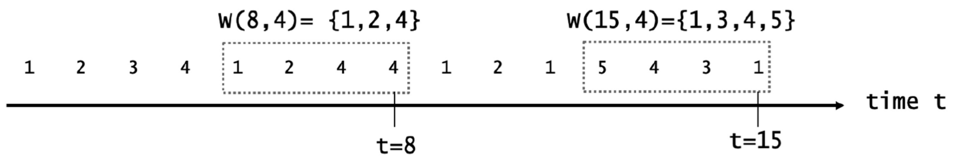


Fig. 5. This example shows the working set of a simple job at two distinct times. We assume time is discrete and each clock tick represents a single page reference. The series of numbers above the time line, called an address trace, is the sequence of page numbers accessed by a job. In this case, there are accesses at times $t = 1, 2, \ldots, 15$. The window size is $T = 4$. The backward window at $t = 8$ contains 4 references to three distinct pages; its size is 3. The backward window at $t = 15$ contains 4 references to four distinct pages; its size is 4. We imagine the window sliding along the time line, giving us a dynamically varying series of working sets.

An important question is how to choose the window size T? When we return to this question later, we will see that we can select a single global value of the window T for which working set management delivers near-optimal system throughput.

Moreover, because the working set policy prevents processes from stealing pages from each other, it is immune to thrashing.

These aspects—simplicity of working set memory management, optimality of throughput, and resistance to thrashing—make working set memory management the ideal of operating systems and caches [13].

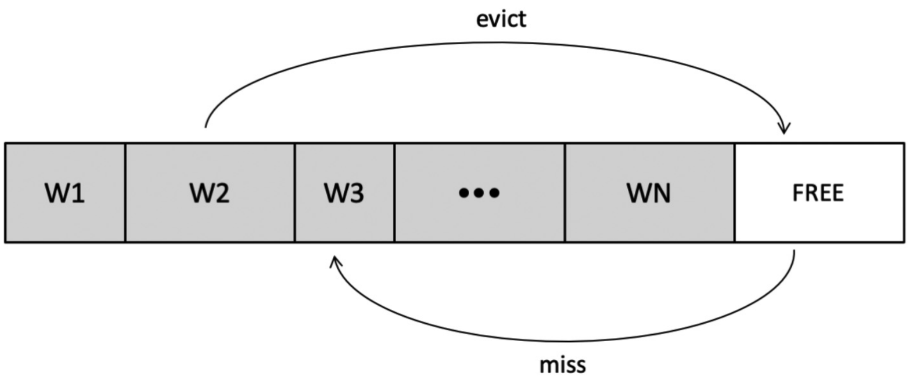


Fig. 6. The box represents memory, a RAM or cache, in use by N executing jobs. Each job's working set is loaded in memory. The FREE area is memory not occupied by working sets. When a job encounters a page miss, the memory slot to hold the new page is transferred from FREE to the job's working set, thus enlarging that working set by one page. (If FREE is empty, then the incoming page replaces the LRU page of the working set.) When a working set page lease T times out, the page is evicted from the working set and the empty memory slot returned to FREE, thus decreasing that working set by one page. When a job quits, all the memory slots it held are erased and returned to FREE. A scheduler maintains a queue of jobs wanting to use memory, admitting the next one only if FREE is large enough to hold its working set. This policy prevents any job from stealing a page from another during a page fault, thereby protecting the system from thrashing.

Working set analytics, discussed next, shows us how to efficiently measure working set statistics to determine the memory capacity of a system and the throughput likely to be observed under a working set policy. The two key statistics are miss rate and mean working set size. All the data needed for these statistics can be measured in a single pass of an address trace and captured as a histogram of reuse intervals. Each statistic is a simple linear computation from those data. The analytics do not depend on any locality or stochastic assumptions about how jobs refer to their pages.

TRACES, REUSES, AND COLD AND WARM STARTS

In this section we will define the basic terminology and notation used throughout working set analytics.

All the statistics are measured in discrete *virtual time*, which is the time of program execution, one tick per memory access. Delays for input, output, or time-slicing are ignored. Measuring in virtual time allows us to see the inherent memory demand of a program without distortion by random interrupts. When needed, we can convert these measures back into real time by inserting delays when interrupts occur.

An *address trace* is a recording of the sequence of page numbers referenced (accessed) by a job at times $t = 1, 2, \ldots, N$; $r(t) = i$ means that the process accessed (used) page i at time t. OS performance analysts use address traces as inputs to simulators of memory management policies in the OS or the cache. Traces can also be used to build page reference maps such as Figure 1. The intervals between successive references to a page are called *reuse intervals*. Figure 7 illustrates.

The miss count, $mc(T)$, is the number of misses—references not in the working set. Misses cause page faults. We occasionally speak of the "miss rate," which is simply the miss count divided by N. It is sometimes suggested that, to prevent the large speed gap of 10^6 between main and secondary

refs:	1	2	3	4	1	2	4	4	1	2	1	5	4	3	1
reuses:	x	x	x	x	4	4	3	1	4	4	2	x	5	11	4

Fig. 7. The address trace of Figure 5 is shown again in top row. It spans 15 time units ($N = 15$) and 5 pages ($M = 5$). The reuse intervals appear just below each reference; a reuse interval is the time since prior reference. The marks "×" indicate first references, which have no prior use. Reuse intervals are important, because they indicate whether a page is in the working set. For example, at time $t = 6$, page 2 is reused with interval 4; a working set with windows 3 or smaller will generate a page miss at that time.

memory from ruining performance, we should choose T large enough, if possible, to keep miss rates no higher than 10^{-6}. As we will see, however, this is not the best way of choosing T.

The first references to pages are different from the others, because they have no reuse intervals. Whether they cause initial page faults depends on how the memory is initialized. There are two possible initializations. The *cold start* (or normal) initialization is simply an empty memory; the first references cause page faults. The *warm start* initialization contains all M pages; the first references cause no page faults. The working set contents throughout the address trace are the same for cold and warm start. The difference is that with cold start every first reference is a miss; with warm start every first reference is a hit. The miss count $mc(T)$ is used for cold start and a new count $mw(T)$ is used for warm start; the difference is

$$mc(T) - mw(T) = M$$

The distinction between cold and warm starts is used in operating systems for the general notion of whether resumption of a suspended job occurs with empty memory or with the previous contents of memory. Cold starts are slow, because all the data must be reloaded from the disks or other sources; warm starts are fast but require a lot of initial memory. The address trace model implicitly assumes warm re-starts of processes suspended by interrupts—the memory contents at time $t+1$ are just what they were after time t, whether or not an interruption occurred between t and $t+1$.

Some analysts speculated that virtual memory performance would improve if the OS achieves warm start by preloading pages. Our formulas, however, show that in the long run there is very little difference between initial cold and warm starts of address traces.

MEAN WORKING SET SIZE

The definition of mean working set size over an address trace is

$$s(T) = \frac{1}{N}\sum_{t=1}^{N} w(t,T) = \frac{st(T)}{N}$$

where $st(T)$ is the space-time occupied by working sets in the trace; one unit of space-time means that one page (space) was in memory for one time unit (time). We see that [8, 10, 11]

- When $t < T$, the T-window extends backward before the start of the trace; only the portion of the window contained in the trace counts toward $st(T)$. Thus, for $1 \le t < T$, the working set size can be written $w(t,t)$.
- For $t \ge T$, there are N-T+1 working sets with window T for the remainder of the address trace.

In some of the analytics discussed below, this distinction is important, and we will separate the original sum into two parts for $t < T$ and $t \ge T$. We will work with four space-time and counting measures:

st(T) = space-time accumulated by working sets of window T
mc(T) = cold-start miss count, the number of misses with window T
mw(T) = warm-start miss count, same as *mc*(T) excluding M first references
mwh(T) = warm-start-hot-finish miss count (defined below)

Each measure can be converted to a time average by dividing by N, the trace length. For example, the classical mean working set size and miss rate are as follows:

$$s(T) = \frac{st(T)}{N}$$

$$m(T) = \frac{mc(T)}{N}$$

COLUMN SUMS, ROW SUMS, AND RUNS

We can depict dynamic memory use with a bit-matrix with one column for each reference for $t = 1, \dots, N$ and one row for each page $i = 1, \dots, M$. Position (i,t) is 1 if page i is in working set $W(t,T)$ and 0 if i is not in $W(t,T)$. Let us call this matrix the *working set residency map*. In the map, the space-time occupied by working sets, *st*(T), is the total number of positions containing 1. Figure 8 illustrates.

Let *col*(t) be the number of 1's in column t of the matrix; notice *col*(t) is the working set size $w(t,T)$.

Let *row*(i) be the number of 1's in row i.

A *run* is a series of consecutive 1s starting with a miss and ending with either a 0 or end of trace. Figure 9 illustrates. The final run may terminate at time N with no more 0's.

LITTLE'S LAW FOR WORKING SETS

Little's law is very useful in queueing analyses, because it gives a relation between three mean values: The mean *number* in a system is the product of the mean *holding time* in the system and the *throughput* of the system. For a working set considered as an evolving system containing pages, the number is the mean working set size, the holding time is the mean length of a run, and the throughput is the miss rate:

$$s(T) = R(T)m(T)$$

This law is easy to verify from our definitions. Because every run begins with a miss, the total number of runs is *mc*(T). The mean run length is

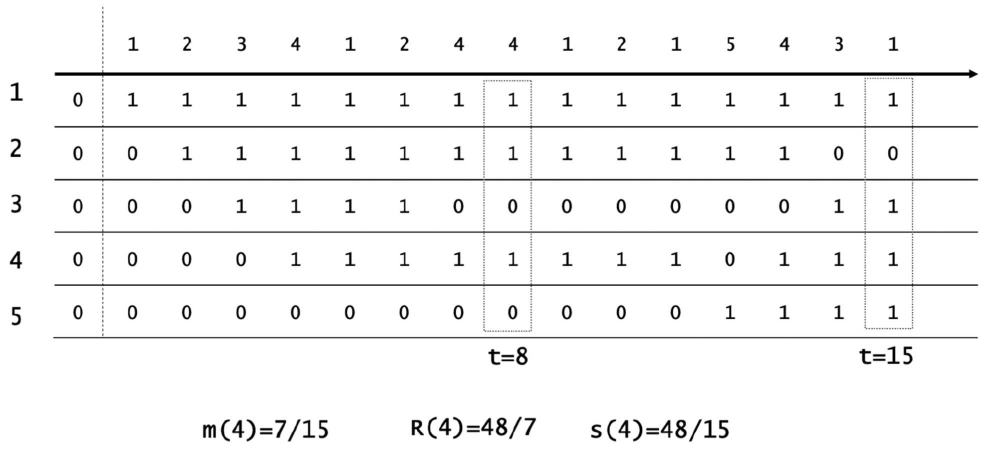
$$R(T) = \frac{1}{mc(T)} \sum_{i=1}^{M} row(i).$$

Then,

$$s(T) = \frac{1}{N} \sum_{t=1}^{N} col(t) = \frac{1}{N} \sum_{i=1}^{M} row(i) = \sum_{i=1}^{M} \frac{row(i)}{mc(T)} \frac{mc(T)}{N} = R(T)m(T)$$

In the example of Figure 8, the total space-time is 48 and number of runs is 7; thus $s(4) = 48/15$, $m(4) = 7/15$, and $R(4) = 48/7$.

Little's law also quantifies the amount of overhang (see Figure 9). For a very long address trace, every run terminates with a reuse interval $> T$ with overhang T-1. Take the "system" to be a delay line of T-1 time units representing the overhang. The throughput is $m(T)$. The response (holding) time of a page in overhang is T-1. Therefore the mean number of pages in overhang is the product $(T\text{-}1)m(T)$. (This argument is not exact, because some overhangs at the end of the trace are shorter than T-1; we will show shortly how to correct for this.) The mean overhang is useful to assess the



		1	2	3	4	1	2	4	4	1	2	1	5	4	3	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0
3	0	0	0	1	1	1	1	0	0	0	0	0	0	0	1	1
4	0	0	0	0	1	1	1	1	1	1	1	1	0	1	1	1
5	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1

t=8
t=15
m(4)=7/15
R(4)=48/7
s(4)=48/15

Fig. 8. The address trace given earlier is shown across the top, labeling the columns, and the pages along the side, labeling the rows. Each map position (*i,t*) is marked with 1 if the page *i* is in the working set ($T = 4$) at that time *t*, or 0 otherwise. The columns represent the different working sets and the number of 1s in a column is the working set size. The column at $t = 0$ indicates that the working set is initially empty (cold start). (Warm start would be represented by a column of 1s.) Note that a miss occurs whenever there is a 1 in a row immediately preceded by a 0. Note also that this is a miniature page reference map with sampling interval 1.

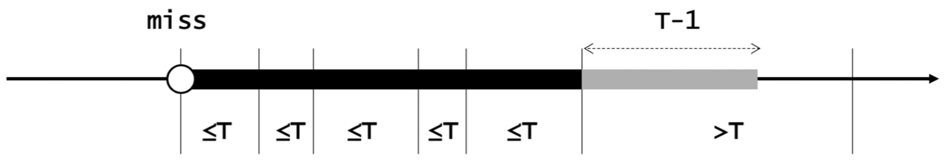


Fig. 9. A run is a time interval during which a particular page is continuously in the working set. In this illustration, each vertical mark is a reference to the particular page and the distances between vertical marks are reuse intervals. A run begins with a miss, contains zero or more reuse intervals $\leq T$, and ends in a reuse interval $> T$. In the reuse interval $> T$, the page remains in the working set for *T*-1 time units beyond the previous reference—the *overhang*. A run can end with a smaller overhang after the last reference to the page: a final interval of length $k \leq T$ houses an overhang of *k*-1, not *T*-1. Each new miss starts a new run.

distance WS is from optimal behavior: if we could eliminate all overhangs, the working set would be optimal. We will prove this shortly.

WORKING SET RECURSIONS

In working set analytics, recursions are simple formulas that relate a measure at window size *T*-1 to the measure at window size *T*. We will derive recursions for miss count and working set space-time. These recursions enable us to calculate miss rate and working set size iteratively in linear time – that is, $O(N)$ for *N* the address trace length.

The algorithm for miss rate is much faster than its counterpart in the Fixed Memory View. The best algorithm in the Fixed Memory View simulates the stack to get the stack distances, requiring time $O(MN)$. This difference can be significant. For example, a 32-bit address space (2^{32}) composed of 4,096-byte pages (2^{12}) has $M = 2^{20}$ (10^6) distinct pages; computing fixed-memory fault count would therefore take a million times longer than the working set miss count.

In a single pass of the trace, we can accumulate a histogram of reuse counts,

$$c(k) = number\ of\ reuse\ intervals\ of\ length\ k$$

for $k = 1, \ldots, N$. We will also include one additional counter $c(x)$ that counts the number of first references. These counters can all be updated in linear time.[8]

By definition, there is no reuse interval preceding the first reference to a page. The next reference is a miss if the reuse interval length is greater than T. In other words, for warm start,

$$mw(T) = \sum_{k>T} c(k),$$

and thus for cold start,

$$mc(T) = M + mw(T)$$

The miss count satisfies the simple recursion

$$mc(T+1) = mc(T) - c(T+1)$$

with the initial condition $mc(0) = N$.

Now we turn to a recursion for the working set size. To do this, we exploit an inclusion property: the runs for window T+1 include those for window T.

What happens when we increase T to T+1? At first approximation, every run is extended by just 1 unit: All the "reuse $\leq T$" intervals are also "reuse $\leq T$+1" intervals and only the final "reuse $>T$" has room for expansion. Its expansion is one unit of space-time. Because the total number of runs is $mc(T)$, the total number of 1's added to the reference map is $mc(T)$. Thus,

$$st(T+1) = st(T) + mc(T)$$

This recursion is exact for infinite address traces [8, 10], but contains an error for finite traces [11]. The errors occur in the end intervals of pages as follows. There are M end intervals, one for each page. An end interval is the time from the last reference to a page until the end of the trace. Notice that this is the same as if we pretend there is another, phantom reuse of page i at time N+1. Notice also that an end interval of length $\leq T$ will not expand when T increases to T+1. To correct for this, we need to deduct from $mc(T)$ the number of end-intervals of length $\leq T$. We can define an end factor

$$e(T) = number\ of\ end\ intervals \leq T$$

and then the accurate recursion is

$$st(T+1) = st(T) + mc(T) - \ e(T)$$

We can express this with the means

$$s(T+1) = s(T) + m(T) - \frac{e(T)}{N}$$

Because $e(T) \leq M$, the end factor vanishes as N becomes large.[9]

We can simplify this by looking closely at the way the end intervals contribute to $e(T)$. There is only one end interval for each page i. Define the end-counter $ec(k) = 1$ if a page has end-interval of length k and 0 otherwise. Then the sum of all the $e(k)$ is M. The last two terms in the st-equation above can be reduced:

$$mc(T) - e(T) = \ M + \sum_{k>T} c(k) - \sum_{1 \leq k \leq T} ec(k)$$

[8]This can be done by maintaining time stamps, $last(i)$, for the last reference to page i. At a reference to page i at time t, the reuse interval is $k = t\text{-}last(i)$. After the last reference, the end interval is $k = N+1\text{-}last(i)$. Adding 1 to $c(k)$ for each of these events leaves corrected values in the counters. This is the same procedure as in the 1978 paper with Slutz [11].
[9]It is interesting to note that $e(T) = w(N+1, T)$, the number of pages in a working set measured at the end of the trace. The reason is that a page is in that working set if and only if its final reference is within the last T window of the trace.

$$
\begin{aligned}
&= M + \sum_{k>T} c(k) - \left(M - \sum_{k>T} ec(k)\right) \\
&= \sum_{k>T} (c(k) + ec(k)) \\
&\overset{\Delta}{=} mwh(T)
\end{aligned}
$$

In other words, the term $mc(T)$-$e(T) = mwh(T)$ is computed by adding the end-correction counts to the reuse counts $c(k)$. We call the miss count with warm start and corrected end intervals the "warm start hot finish" miss count. It is a warm start, because initial page faults are not counted. It is "hot finish," because the end corrections are applied at the very end by pretending that all M pages are simultaneously referenced at time N+1. This can be summarized as the working set size recursion:

Working Set Size Recursion

$$st(T+1) = st(T) + mwh(T)$$

where $st(T)$ is the space-time accumulated by working sets of window T and $mwh(T)$ is the warm start hot finish miss count. The initial conditions are $st(0) = 0$ and $mwh(0) = N$

Notice that for very long address traces (large N), the M end corrections are insignificant compared to the total of all the counters. For large N, $mwh(T)$ converges to $mc(T)$.

The recursion we reported 1972 was mathematically the same, but it depended on the assumption that the reference string is a random (stochastic) process that enters a long-term steady state [10]. In 1978, we removed the stochastic assumption and found the recursion works for finite address traces if we make end corrections to the counters. The working set recursion stated here has a much simpler derivation.

The working set recursion can also be run "backwards" to deduce the miss rate given the mean working set size. If we had a direct measurement of the mean working set size, then we could find the miss counts by taking the differences $mc(T) = st(T+1)$-$st(T)$ [32].

For completeness, we summarize the miss rate recursions:

Summary of Miss Rate Recursions

Miss rate set recursions enable the linear-time calculation of miss counts at window size T+1 in terms of those for window size T. For miss counts:

$$mc(T+1) = mc(T) - c(T+1)$$

where $c(T)$ is the count of reused intervals of length T, with initial conditions $c(0) = 0$ and $mc(0) = N$. For warm-start-hot-finish

$$mwh(T+1) = mwh(T) - cc(T+1)$$

where $cc(T) = c(T)+ec(T)$ is the count corrected for end intervals. The initial conditions are $cc(0) = 0$ and $mwh(0) = N$.

The miss count measures mwh and mc both contain N counts; mc includes M first references but no end corrections, and mwh includes no first references but has M end corrections. As N gets large, both rates $mc(T)/N$ and $mwh(T)/N$ converge to $m(T)$.

Uses:	1	2	3	4	1	2	4	4	1	2	1	5	4	3	1
Reuse intervals:	x	x	x	x	4	4	3	1	4	4	2	x	5	11	4
End intervals:	x	x	x	x	x	x	x	x	x	6	x	4	3	2	1

Fig. 10. The top row is the address trace from Figure 7. The reuse intervals are shown in the middle row, where each "×" marks a first reference. The end intervals are shown in the bottom row, where each "×" marks a non-final reference.

Table 2. Statistics of the Example Address Trace

k	c(k)	ec(k)	mc(k)	mw(k)	mwh(k)	st(k)
0	0	0	15	10	15	0
1	1	1	14	9	13	15
2	1	1	13	8	11	28
3	1	1	12	7	9	39
4	5	1	7	2	3	48
5	1	0	6	1	2	51
6	0	1	6	1	1	53
7	0	0	6	1	1	54
8	0	0	6	1	1	55
9	0	0	6	1	1	56
10	0	0	6	1	1	57
11	1	0	5	0	0	58
12	0	0	5	0	0	58
13	0	0	5	0	0	58
14	0	0	5	0	0	58
x	5					

EXAMPLE

Figure 10 shows the sequence of reuse intervals and end intervals for the example address trace. We tally these data along with the miss counts and working set space-time in Table 2. The row for $k = 0$ gives the initial conditions.

These data are plotted in Figure 11 and compared with LRU and MIN. The graph shows a region at the smaller memory allocations where LRU is better than WS. A similar pattern is sometimes observed in actual programs [29, 30].

OPTIMAL MEMORY POLICY

Now we will examine the optimal policies for managing memory. The optimal policy for Fixed Memory is MIN. It was defined by Les Belady in 1966 [2].[10] MIN's principle, *invoked at each page fault*, is "replace the page that will not be reused for the longest time in the future." This policy is unrealizable because the operating system cannot know the future. However, its fault count can be computed in a single pass of an address trace with about the same overhead as for LRU: order $O(NM)$ [23].

[10]Many people find the MIN replacement rule to be obvious. Yet a formal, definitive mathematical proof of its optimality is difficult [1].

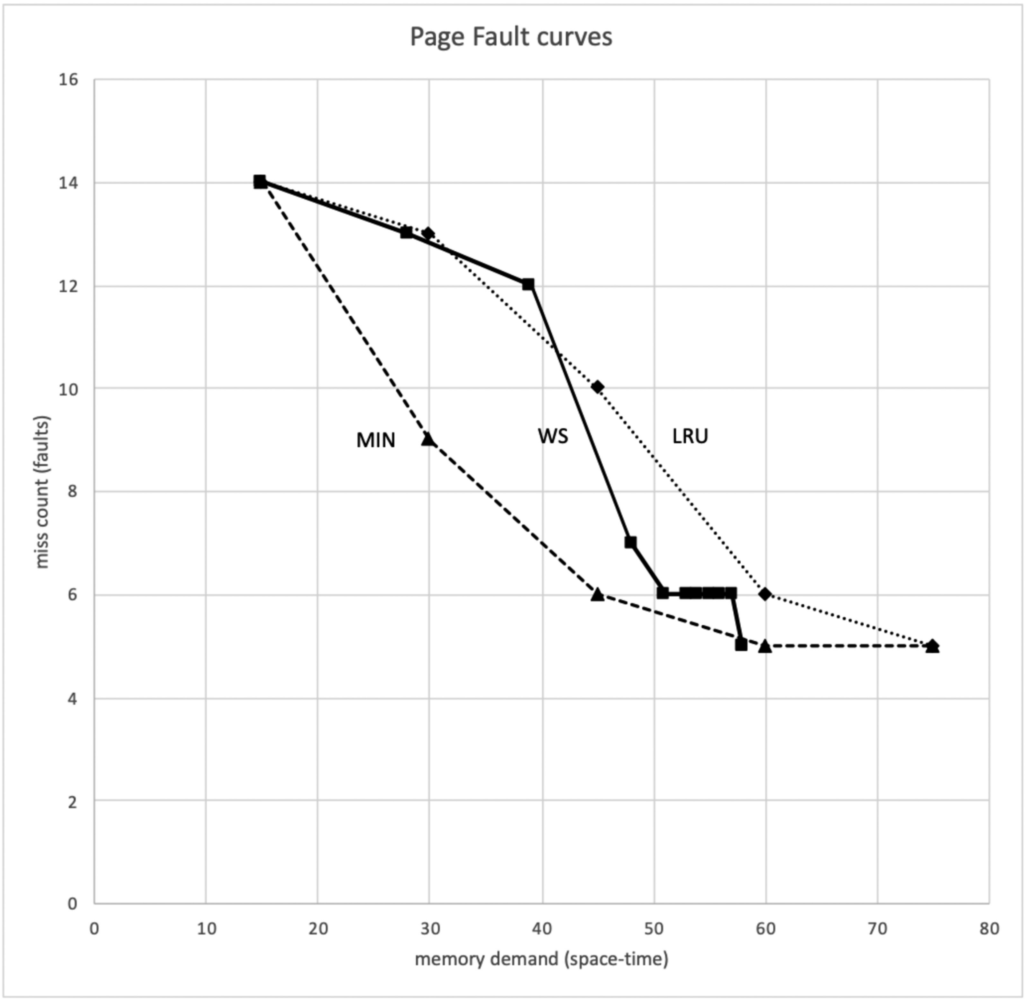


Fig. 11. This graph compares three fault curves. A fault curve plots the number of faults (vertical axis) versus memory demand (horizontal axis). The solid curve is the WS example of Figure 8. LRU and MIN are also shown. For a fixed-space policy, the memory demand in space-time is the product of the memory size and address trace length; here the allowable memory sizes 1, 2, 3, 4, 5 for LRU and MIN correspond to space-times 15, 30, 45, 60, 75. In contrast, WS is variable space and is defined at points corresponding to non-integer average memory sizes. For this address trace, WS is mostly better than LRU and mostly worse than MIN, although at the largest window size, WS is slightly better than MIN. This is not an anomaly, but rather the consequence of working sets being variable in size.

The optimal policy for Shared Memory, in which the space allocated to a job can vary, is VMIN. It was defined by Barton Prieve and Robert Fabry in 1976 [25]. VMIN's principle, *invoked after each reference*, is "retain the current reference in memory if its forward reuse interval is $\leq T$, otherwise delete it immediately." This policy is also unrealizable. However, its fault function and mean size can be calculated rapidly as for WS: order $O(N)$.

It is interesting that database designers discovered the same optimizing principle in the 1970s [17]. Here is their argument. Suppose we want to decide when to keep a page of data in main memory versus the much slower hard disk. Just after the page is used, we look into the future to see exactly when it will be used again. Then we do a simple calculation to compare the rental cost of keeping it in memory until next use with the cost of removing it immediately from memory and paying the swapping cost for its retrieval later. The database designers found that with typical parameters for memory cost and disk delay, a data page should be deleted if it has not been reused after about 5 minutes.

VMIN uses the window of size T as the threshold point at which retaining until next reuse costs the same as a page fault. To state this precisely, suppose time $t+x$ is the next reuse of the page referenced at time t; VMIN decides whether to retain that page or not as follows:

If $x > T$, then *immediately* evict the page from memory;

If $x \leq T$, then *keep the page continuously in memory* until next reuse.

VMIN exercises this choice at every time t. Because each and every reference is evaluated for minimum cost, the total cost must be minimum, too.

VMIN is just like WS, but with a forward looking rather than backward looking window [13]. If the page is used again in the forward window, then VMIN retains it in memory until next use. If the page is not used again in the forward window, then VMIN immediately removes it from memory. VMIN's page fault sequence is identical to WS.

When the reuse interval x is $\leq T$, WS and VMIN retain the page continuously between the two successive references. If $x > T$, then WS retains the page for additional time until evicting it, whereas VM evicts it immediately. This means that the space-time difference between WS and VMIN is due entirely to the overhangs in the reuse intervals longer than T. This observation allows us to calculate the VMIN space-time $vt(T)$ directly from reuse counters. Start with $vt(1) = N$, because with window size 1 only the current references are in the VMIN memory. A reuse interval of length $k \leq T$ contributes an additional k-1 to the space-time. Thus,

$$vt(T) = N + \sum_{k=1}^{T} (k-1)\, c(k)$$

We get a recursion straightway,

$$vt(T+1) = N + \sum_{k=1}^{T+1} (k-1)c(k) = N + \sum_{k=1}^{T} (k-1)c(k) + Tc(T+1)$$

or, simply,

$$vt(T+1) = vt(T) + Tc(T+1)$$

Therefore, as with WS, the VMIN space-time and miss rate can be calculated directly from the reuse interval counters $c(k)$, without a simulation of VMIN.[11]

We can calculate the difference of WS and VMIN space-time simply as the total overhang in reuse intervals $> T$. Specifically, the overhang is T-1 in any reuse or end interval $> T$; by our previous calculations there are $mwh(T)$ of these. For all reuse intervals of length $k \leq T$, there is no overhang, but end intervals of length $k \leq T$ have an overhang of k-1. This leads to the relation

$$st(T) = vt(T) + (T-1)mwh(T) + \sum_{k=1}^{T} (k-1)\, ec(k)$$

[11]By comparison, the WS space-time recursion has a miss-rate term $mwh(T)$, the sum of counters, instead of a single counter $c(T+1)$. That is because when we increase T to $T+1$, the WS overhang grows in *all* the reuse intervals $> T$, the total number of which is the sum of *all* the counters $c(k)$ for $k > T$. VMIN is parsimonious: increasing T to $T+1$ only adds the space-time of reuse intervals of *exactly* length $T+1$.

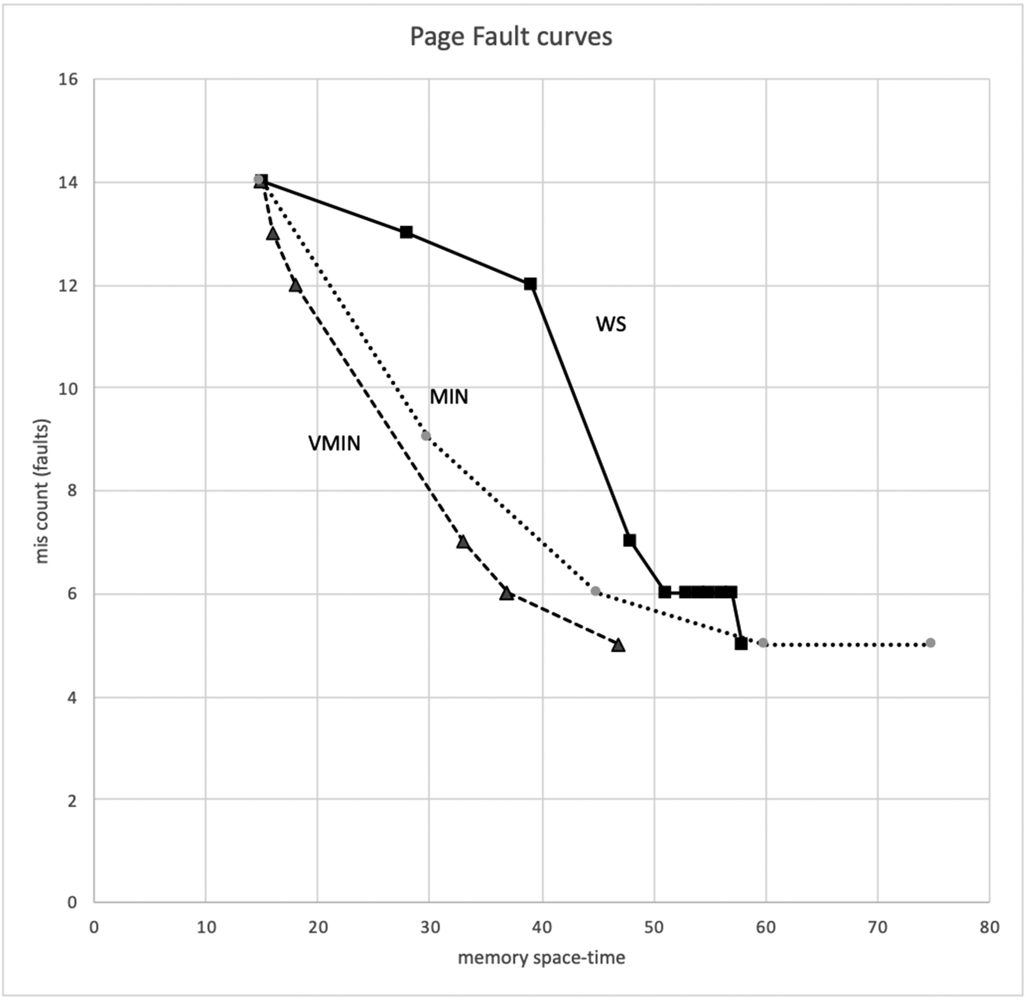


Fig. 12. This graph plots the VMIN space-time and the WS space-time. The MIN curve from the previous figure is included for comparison. If we pick any value of miss count, then we can compare the VMIN and WS memory demands. For example, at miss count 8, VMIN space-time is 30 page-units and WS space-time is about 47; the VMIN memory demand is about 2/3 of the WS demand for the same paging rate. Notice that the VMIN curve is always convex, whereas WS is not. WS and VMIN are not close in this example, because the address trace is too short to exhibit locality.

Figure 12 illustrates this for the data of the previous table.

The lesson from all this math is that we can quickly compute the space-time of a VMIN policy from the same reuse interval statistics as for WS.

GOOD LOCALITY MAKES WORKING SET NEAR OPTIMAL

I would like to clarify a point about the assumptions behind the mathematical formulas given above. The *only* assumption is that an address trace can be recorded from a computation. The recursion formulas for fault rates and working set size are simply mathematical relationships for measures computed on address traces. Optimality defines the best that can be done on a given address trace. There are no assumptions about locality or stochastic steady-state built into the working set and optimality measures.

The page reference map is also constructed from an address trace. Locality appears in the distinctive phase-transition patterns of the map. The working set was devised to measure the locality sets seen in the map. It makes no assumptions about locality.

Locality comes in when we claim that WS is close to the optimal VMIN. Let's see how that works with the reference map in Figure 1.

Consider a locality set of P pages and its associated phase consisting of L sample windows of length T. Each sample window within the phase contributes PT space-time. VMIN and WS have the identical memory contents everywhere in the phase except for the first and last sampling windows. In the last window, WS will have PT space-time, but VMIN will have only a fraction $h \geq 0$ of PT, because it unloads pages after their last references before the phase ends. In the first window, WS will have an increment of space-time from overhang of pages from the previous locality set; this increment can be expressed as FPT, where F is the ratio of the size of previous locality set to the current one. Also in the first window, VM will incur a fraction $g \geq 0$ of PT, because it loads the new pages incrementally. With these considerations, the ratio of the two space-time measures is

$$\frac{st(T)}{vt(T)} = \frac{FPT + (L-1)PT}{gPT + (L-2)PT + hPT} < \frac{L+F-1}{L-2}$$

The inequality results from replacing g and h with their lower bounds (0) in the denominator.

Figure 1 shows 5 locality sets and their associated phases for a sampling window T of approximately 380K units. A measurement of the map to find the locality set sizes and durations yields the Table 3, where size is the number of pages in the locality set and length is the number of sample intervals in a phase.

The final column in the table above shows the bounding ratio for each of the phases of Figure 1. For the longer phases, VMIN and WS are about 1% apart. For the short phase, they are about 3% apart. A more precise analysis would have narrower separations than this approximate analysis.

VMIN's virtual space time is less than WS's. How does this translate to real space-time as needed by the memory space-time law? From our previous analysis of the space-time law, working set real space-time is estimated as $st(T)(1+Dmc(T))$, where D is the ratio of disk to RAM time. Since WS and VMIN have identical paging $mc(T)$, the estimate for VMIN real space-time is $vt(T)(1+Dmc(T))$. Thus VMIN's minimum virtual space time translates directly to minimum real time space-time and maximum throughput. This is why when WS is close to VMIN, their respective system throughputs are close.

In summary, working set analytics provides tools for measuring locality. When locality is present, a working set memory management policy will set system throughput to within a few percentages of optimal.

COMPARISONS

Memory policies can be classified in two dimensions: fixed or variable space, optimal or not. We have shown representatives of each combination in Table 4.

The optimal policies are generally not realizable in real time, because they require knowledge of the future reference patterns. Although the optimal policies are not realizable, their performance is easy to compute from a recorded address trace. Therefore, it is easy to measure how far a realizable policy (LRU or WS) is from optimal.

Fixed space policies are used for memories of fixed size such as caches or hard partitions of memory in an operating system. Fixed space policies are subject to thrashing when extended to shared memory or shared cache, because the space allocated to every job decreases as the number

Table 3. Comparison of WS and VMIN

set	size	Length *L*	*F*	(*L*+*F*-1)/(*L*-2)
1	50	180	0/50 = 0	1.01
2	65	220	50/65 = 0.77	1.01
3	30	220	65/30 = 2.2	1.01
4	55	50	30/55 = 0.55	1.03
5	35	180	55/35 = 1.6	1.01

Table 4. Taxonomy of Memory Policies

	Fixed space	Variable space
Not optimal	**LRU**	**WS**
Optimal	**MIN**	**VMIN**

of jobs increases. The WS policy (Figure 6) is resistant to thrashing, because it cannot steal pages from other processes and because the scheduler will not load new processes if their working sets do not fit into the available free space of memory.

What happens when these policies are applied for jobs displaying a strong degree of locality? When the memory allocated by the policy is too small to hold the job's locality sets, LRU and WS would be comparable—but with poor performance. When memory is sufficient to hold locality sets, LRU and MIN would be comparable, and WS can be better than both, because the fixed-space policies retain pages not being used in a locality set.

Moreover, LRU is also susceptible to thrashing when extended to shared memory. WS will not thrash and it is close to optimal.

CHOOSING THE WINDOW SIZE

What is a good window size? Is there a best window size for each process? What would happen if all processes were measured by the same window size?

This question has been investigated by researchers dating back to the 1970s. One basic finding is that graphs of WS space-time versus T show that the minimum of $st(T)$ is in a wide, near-flat valley in most jobs. There is usually a single value of T that intersects all the vallies of the jobs. In other words, it does not make much difference what value of T you choose; T can be a single, global value chosen over a wide range without significantly changing the system throughput.

From the page reference maps, we see that an ideal T is just large enough to see all the locality pages in the window throughout the phase. In Figure 1, for example, $T = 380$K is sufficient to see all the pages of the locality set. That value of T is a small fraction of the phase length—less than 0.5% for the four long phases and 2% for the one short phase.

This gives a practical answer to the performance tuning problem mentioned at the beginning. A system administrator can adjust the parameter T in a WS-managed system to find a value that maximizes system throughput. After that, the parameter T does not need to be changed. That maximum will be close to the theoretical optimum of VMIN.

IMPLEMENTATIONS

There are several ways to implement the working set cheaply.

The original ideas for implementation (1968) were of two kinds [8]. The first was to have the operating system scan and reset the use bits in page tables every T time units of virtual time, removing any unused pages from working sets. This implementation was not attractive, because scanning all the pages tables every T time units in a large system would be very expensive.

The second idea was to associate a hardware timer of duration T with each page frame of memory. The timer is set to T at each access to that frame. It ticks down to 0 after T time units of non-use. The operating system can detect these unused frames from their expired timers and remove them from their working sets. This implementation was not attractive at the time, because the required hardware would be too expensive, and, moreover, because of CPU multiplexing, the timers would need to be shut off when the job owning the page was not running.

Chen Ding and his students at University of Rochester note that in modern caches, multiple CPUs execute in parallel and share on-chip L1 and L2 cache. The hardware is easily designed to include timers on cache pages. Because there are multiple cores, there is no need to shut any timers off to accommodate round-robin CPU scheduling. Their proposal of a "lease cache," in which each cache page has its own timer, is now feasible [20]. A lease cache with lease T is exactly the working set policy with window T.

A simple modification of the popular CLOCK algorithm enables the operating system to detect the working set pages of a job. As before, the process's pages are arranged in a circular list, but now each page has a time stamp instead of a use bit. The time stamp records the time of the last access to the page. The scanning clock hand skips over all pages whose time stamp is within T of the current time, and stops at the first page with an older time stamp. This method, called WSCLOCK, was invented and validated by Richard Carr [7].

Note that WSCLOCK has no built-in load control. By itself, it cannot implement the full WS multiprogramming policy (Figure 6), which does not load another job until there is sufficient free space for its working set. However, the WSCLOCK search time for a timed-out page can be a proxy for the measure of free space: the longer the search time, the less free space is available. The memory allocator can load a new job when the search time is below a set threshold. (Note that the memory allocator is not the same as the CPU core scheduler. The memory allocator decides when to load a job's working set into memory. The scheduler assigns free CPU cores to jobs whose working sets are loaded.)

APPLICATION IN MODERN CACHES

Modern computer chips rely on complex hierarchies of caches to achieve and maintain high performance. Typical cache levels L1, L2, and L3 buffer the gap between the RAM and the CPU. Level L1 is closest to the CPU and is the smallest and fastest of the caches. There are two kinds of cache configurations. Inclusive cache means that cache pages (also called slots or blocks) of a level are subsets of larger pages at the next level down. Exclusive cache does not require this subsetting [34]. These caches usually use some form of LRU replacement to determine when a cache page is pushed down to the next lower level cache. The highest-level cache (L1 and L2) is dedicated to individual CPU cores, while L3 cache is shared among all the cores similar to multiprogramming [34]. The shared L3 cache may be partitioned unequally, with some cores getting more than others depending on their locality [4]. Researchers are investigating whether a variable partition based on CPU core working sets would be more efficient.

Chen Ding's research group looks deeply into policy and performance questions for shared caches [21, 31, 32, 33]. Many of their analytic methods for caches were inspired by the working set theory. They defined two measures of load on the cache, called *footprint* and *footmark*. Let us discuss both of them and compare with WS.

The footprint measure was motivated by a desire for accuracy. For the first T-1 references of a trace, the working sets are the contents of a truncated window, because there are no page references prior to time t = 1. In fact, the initial working sets for the first T-1 references are of sizes *w*(*t,t*). Those truncated windows present a potential underestimate in the calculations of mean working set size. Chen Ding's *footprint* avoids these complications by averaging together all the working sets whose *T* windows fit completely inside the address trace:

$$fp(T) = \frac{1}{N - T + 1} \sum_{t=T}^{N} w(t, T)$$

Ding argued that the derivative of the footprint function is the miss rate, just as indicated by the working set recursion. Thus, the miss rate can be computed once the footprint is computed. His team found a recursion for computing *fp*(*T*) from *fp*(*T*-1) and showed that the mean footprint is within one page of the mean working set if the window is not too large:

$$s(T) - fp(T) < 1,\ if\ T \leq \sqrt{2N}.$$

This will be easily true for programs with good locality. Details are in the Appendix.

Chen Ding was not happy with the messiness of algorithms for computing footprint. He searched for a recursion based on footprint that was identical to the one reported by Denning and Schwartz for long address traces in stochastic steady state. He and his students found a new, closely related measure they called *footmark* [35]. Footmark satisfies the Denning-Schwartz recursion,

$$fm(T + 1) = fm(T) + m(T)$$

with initial conditions $fm(0) = 0$ and $m(0) = 1$. The details of its derivation are in the Appendix.

We now have two recursions—working set size and footmark—that satisfy the same mathematical form,

$$F(T + 1) = F(T) + M(T)$$

where *F* is a space measure and *M* a misses measure. The working set uses the warm-start-hot-finish miss rate *mwh*(*T*) for *M* and footmark uses the actual working set miss rate *mc*(*T*) for *M*. For finite address traces, the two miss rates differ by a few end corrections. But for very long traces (large *N*), they become identical and the two recursions are the same. That replicates the 1972 finding for the steady-state values of working set size *s*(*T*) [10]. Like footprint, the footmark measure is within one page of the mean working set size when $T \leq \sqrt{2N}$. Figure 13 illustrates these measures for the example address trace used previously. When there is a risk that $T > \sqrt{2N}$, the best strategy is to use the working set recursion, which is fully accurate and is easily computed from *mwh*(*T*), which in turn differs from *mc*(*T*) by a few end corrections.

WORKING SETS IN PARALLEL ENVIRONMENTS

The computing environments in which virtual memory was formulated featured a single CPU accessing a memory shared by many jobs. It was basically a serial environment. The CPU of old has evolved into modern multicore chips, graphics processors (GPUs), and several kinds of cache. Does all this parallelism invalidate any assumptions of the analytics? Do graphics and neural networks, the primary applications of GPUs, exhibit locality?

As for the first question, the assumptions of working set analytics do indeed apply in a parallel environment. The analytics depend only on the assumption that an address trace can be measured. A particular run of a parallel system yields an address trace for that run. The next run of the same system may yield a different address trace because the order of events is different or the input data are different. But this does not create any problems for the analytics, because the analytics are defined for a given address trace. WS will reveal the locality sets of that trace and adapt to them.

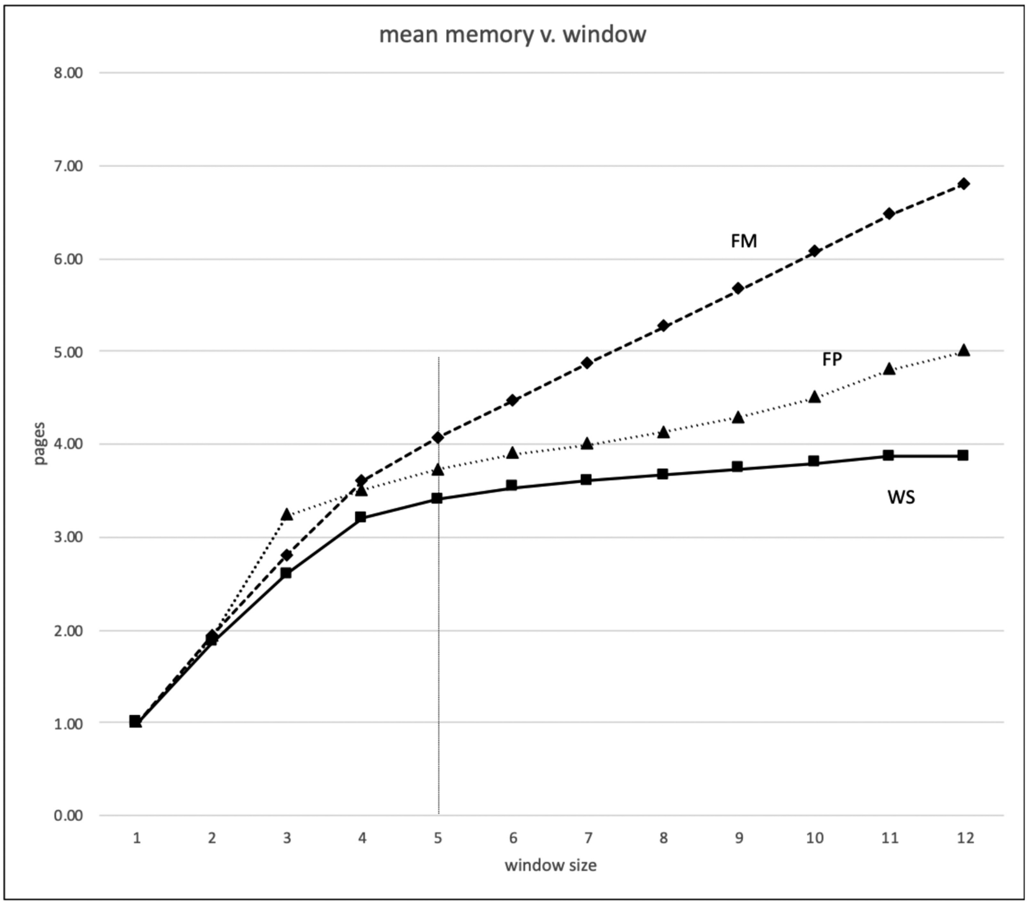


Fig. 13. This graph compares the footprint (FP), mean working set (WS), and footmark (FM) measures for the 5-page example address trace. The working set rises asymptotically toward 4.0 pages. Both the footprint and footmark continue growing. When $T \leq \sqrt{2N} = 5$ (vertical line) footprint and footmark are within 1 page of WS as predicted by the bound. For example, in the page reference map of Figure 1, there are over 1,000 sample intervals; T could be as large as 45 (= $\sqrt{2000}$) sample intervals without causing significant error between footmark and mean working set. When N is very large (not shown in this diagram), the two measures FM and WS converge to the same.

VMIN will reveal the best possible performance for that trace. What happens in other traces does not affect the analytics.[12]

As for the second question, consider that a system is a set of CPUs accessing a set of pages in memory. Whenever any CPU, running any job, accesses a page, the page's lease is set to T and counts down toward zero with each clock tick. The protocol for deciding whether to keep a page in memory is the same whether or not jobs share pages or run in parallel.

[12]In his book *Rethinking Randomness*, Jeff Buzen discusses how many aspects of computer systems can be represented as event traces [6]. The traditional algorithms of queueing theory for utilization, throughput, and response time work for these traces based purely on what can be observed in the trace. The address trace and working set analytics are of this kind.

Whether the lease rule optimizes system throughput depends on the mix between parts of jobs with good locality and parts without. There is lots of debate around this issue. A GPU working with streaming data (for example, running a graphics display) or with simulation of a neural network (doing the linear algebra calculations for each layer) may not exhibit phase-transition behavior. But the rest of the computation, such as feedback for reinforcement learning in a neural network and the user interface, is likely to exhibit good locality behavior. This means that locality is important for optimizing the performance of the non-GPU components and managing communication between job-components running on conventional CPUs and those running on GPUs. Other optimizations, such as streaming caches, which release blocks immediately after begin used, can be applied to the GPU parts.[13]

WIDER APPLICATION OF WORKING SET PRINCIPLES

The principles of memory management in the working set model have been used outside the traditional operating systems that hatched them. The two most prominent are content delivery in the Internet and management of inventories and logistics networks.

Start with content delivery. The Internet hosts major services that distribute data throughout the world. Examples are distribution of music, video, books, and cloud services. Even though these services look like a single entity, they are built as distributed networks, because congestion at a single server would be too great for acceptable service. The networks include caches located in busy zones, such as major cities, so that users in those areas can access the content via physically short high bandwidth paths. Content is automatically downloaded to a local cache when a user of requests new data. Data that have been resident for a long time are automatically refreshed at intervals.

These caches are often called "edge caches," because they represent moving data away from centralized locations that are easily congested, to the edge of the network where the users connect. Numerous companies employ edge caches as part of their content delivery services.

An edge cache functions in the same way as a cache inside the operating system. When a block of data is accessed for the first time, it is loaded into the edge cache from a central server. An LRU replacement policy or a lease policy is used to remove old data when the cache is full. These caches accumulate working sets and achieve high performance by keeping the working sets physically close to their users. All caches perform well by exploiting the principle of locality in the patterns of how users access data.

Turn now to inventory management. An inventory is a set of items or parts. A logistics network is a network of depots that hold inventories of parts used by companies or militaries. When applied to inventories, a working set is a set of parts that have been used recently. Depot managers aim to keep local depots filled with the most needed working parts so that they can supply their local users rapidly. When someone asks for a part not in the local depot, the manager must send a request to another depot to obtain the part.

Given the prevalence of locality behavior, it is reasonable to expect user access patterns for inventories to exhibit locality sets (subsets of possible parts) and phases (intervals of heavy use of a particular locality set). Working set analytics could be used to determine the capacity needed of an inventory depot.

There is a difference between a "logistics working set" and a "memory working set." The logistics working set includes multiple instances of a part, whereas the memory working set has just one instance of a page. The number of requests for a particular item in the working set window

[13]The address trace of a data stream looks an increasing sequence of page numbers with no repeats. It has no locality. VMIN will remove a page from memory immediately after it is used.

forecasts the quantity of that item to keep on hand. It would be worthwhile to study this idea of logistics working sets and their phase transition maps to see what inventory management strategies give optimal performance of the logistics network.

MEMORY MISCONCEPTIONS

Before closing this tutorial, I would like to comment on four common misconceptions about computer memory. They result from unfounded assumptions that lead to pessimistic answers to four big questions:

(1) Is memory flattening?
(2) Is virtual memory obsolete?
(3) Is computer memory irrelevant in the Cloud?
(4) Is the principle of locality obsolete?

Is Memory Flattening?

Memory is an essential component of computers. Most people think of memory as a companion technology that the CPU interacts with to fetch instructions and access data. Performance of the computer then seems to be governed by the CPU speed.

But this is not so. Memory is not a single component as is a CPU chip. Memory is a system that includes caches in the CPU, RAM, disks, network services, and Cloud storage. Whereas a CPU core runs only one job at a time, memory is shared among many jobs. Memory contention begets queueing at high-demand servers. Disk and network bottlenecks at these queues will kill performance of memory systems.

A thought experiment highlights this basic reality about memory. A lot of people believe the claim that within a few years we will know how to make memories that are essentially infinite and flat. Flat means that the access time of any item in the memory is approximately the same. Flat memory would not need locality optimization, because every page would have the same access time. But flat memory is not a thing of the future. It is already here—the Internet. If you issue a "ping" command from anywhere in the Internet for any IP address in the Internet, then you will see that the packet round trip times are mostly 30–90 ms. That is not perfectly flat but is close. Does that mean your computer will experience a maximum delay of 90 ms when it reaches out to a web server? Of course not. If the server is very popular, then many computers all over the Internet will be trying to access it. The server will queue up the requests, because its disks are a bottleneck. The greater the demand for the popular web server, the longer the wait time in the queue—and the longer the response time to get a web page. In other words, flat memory systems do not guarantee good performance. This is why the Internet contains so many edge caches—they spread the load and avoid the queueing.

The same problem appears at smaller scale when many jobs share RAM together and contend for use of the paging disks. Working set memory management prevents disk overload and protects the system from thrashing.

The control systems that prevent excess delay in accessing shared memory resources are an important part of the memory story. Most users and programmers are not aware of those control systems.

Is Virtual Memory Obsolete?

Virtual memory was invented to overcome the high programming cost of solving the "overlay problem"—manually planning page transfers that overwrite previously loaded pages. Today's memories are usually large enough to contain an entire program and its data. With only a few exceptions, virtual memory is not useful any more for solving the overlay problem.

Virtual memory's real value comes from its ability to partition memory. It allocates nonoverlapping subsets of pages to each job's memory. Because a job's page map table does not see the pages of any other job, virtual memory prevents any job from accessing data in another job's memory. Virtual memory provides the basis to encapsulate untrusted software so that it cannot damage anything outside its allocated memory space. Virtual memory provides basic access control by distinguishing read, write, and execute permissions for individual pages. In other words, virtual memory implements the basic guarantees of an operating system for data protection.

Virtual memory does more than partition memory and provide basic access control. It also manages multiprogrammed RAM to avoid thrashing and to maximize system throughput. The same concerns for stability and throughput now occur in the cache on the CPU chip, which is shared among the multiple CPU cores on the chip. The principles of virtual memory are important in cache design.

Is Computer Memory Irrelevant in the Cloud?

The Cloud provides very large storage in the network outside your computer. It does not include the storage inside your computer. Your computer still needs L1, L2, L3 cache, local RAM, and local storage. The operating system on your computer needs to manage these local memory resources. The Cloud enlarges the storage accessible to your computer but does not replace storage management within your computer.

The Cloud is a complex, distributed storage system. It consists of many data centers around the world. Each data center includes tens of thousands of computers and disks. Sophisticated redundancy controls manage multiple copies of files distributed across multiple data centers, providing high reliability and ease of recovery from failures of computers and disks. The operating systems running the Cloud must also manage memory, avoid thrashing, and maximize Cloud throughput.

The Cloud is not the final answer to storage management. One of the primary limitations on performance of computers is the interfaces between CPUs and memory system. Moving data across those interfaces significantly slows CPU speeds. The Cloud is another interface. Those interface delays are often called von Neumann bottlenecks, because the separation of CPU and memory was a central principle of the stored-program computer architecture that became ubiquitous after 1945. Many research projects today are examining new architectures than have no von Neumann bottleneck. The idea is to build the computer so that computations are performed by huge numbers of processing elements that require only local access to limited storage. These are often called "processing-in-memory architectures." Neural networks are a prominent example.

Is the Principle of Locality Obsolete?

The heart of all the methods to control memory and optimize its performance is the principle of locality. Each program generates a unique footprint of locality sets and phases. High performance caches, RAM-DISK interfaces, and networks all owe their success to this principle.

The locality principle runs deep in computing. Algorithm theorists have shown that a procedure cannot be an algorithm unless each of its operations can apply only to a bounded, local set of data. Computing machines themselves are built from many components and modules that use only local inputs.

Yet the locality principle for memory is often misunderstood. Some people believe it simply means unequal frequencies of use of each page. Others believe it means a slow drift among a set of favored pages. Few see it as long phrases of near constant locality sets punctuated by sharp transitions to new locality sets. Yet the phase-transition behavior is common and is the reason why working set is able to give near optimal system throughput.

Another misunderstanding is the belief that incentives have changed. In the early days, programmers of early virtual memory systems purposely induced locality behavior to get the best performance from the paging algorithms. We are in a different age now: Memory is nowhere near as scarce and programmers feel no pressure to induce working sets into their programs. Thus, it would seem that the motivation for locality is gone. This misunderstanding is refuted by two facts. One is the experimental studies showing that locality behavior is present in the source code of programs—the phase-transition behavior appearing in page reference maps is an image of source locality. Locality is the consequence of our problem-solving strategies such as iterative loops and divide-and-conquer. The other refutation is recent instrumentations showing page reference maps with even more pronounced phase-transition behavior than we saw in the early days. The increased use of modular programs is the most likely explanation. (See McMenamin's study of Linux programs [24].)

CONCLUSION

The working set model for program behavior, first articulated in 1965, has stood the test of time. It stimulated the research that revealed that almost all executing processes exhibit locality, establishing the principle of locality as one of the fundamental principles of computing. It led to a simple, precise way to measure all the working set statistics in real time by recording the reuse interval statistics of an address trace. It led to the conclusion that WS is near-optimal for processes exhibiting locality. It provided an explanation for the mystery of thrashing and showed how to build a control system to avoid thrashing.

Since 1965, computer CPUs have become progressively faster, widening the cost gap of retrieving a page from a lower level of memory. At the same time memory has become far cheaper so that most routine applications are fully loaded in memory and do not normally cause page faults. Some people have asked whether the theory applies to real systems today.

Yes, it does. Real systems today have memories consisting of multiple levels of cache, RAM, and disk. The caches near CPU (L1 and L2) are shared among multiple cores (CPUs). Most of the current cache management strategies do not vary the partition of the cache among the cores, but researchers have been demonstrating that variable partition caches are much more efficient. Unfortunately, variable partition LRU caches are susceptible to thrashing. The WS theory can be helpful to design cache management strategies, such as lease cache, that do not thrash and maintain cache misses to close to the optimal levels.

Despite the tremendous advances in memory technology over the past half century, the basic assumptions behind memory management have not changed. Virtual memory remains useful because of its ability to confine jobs to their limited regions of memory, to encapsulate untrusted software, and to manage load to avoid thrashing. Flat memory does not eliminate the need for virtual memory; it introduces its own problems due to queueing and congestion as many jobs access shared memory resources. The Cloud augments but does not replace memory management on local computers. Moreover, the Cloud exacerbates the von Neumann bottleneck between CPU and memory. The locality principle is far from obsolete – it continues to underpin high performance memory systems. The working set theory can be extended and combined with network caching theory for possible application in logistics networks and inventory management.

Working sets and virtual memory will be parts of computing for a long time to come.

APPENDIX: FOOTPRINT AND FOOTMARK

Chen Ding at the University of Rochester defined two measures of load on the cache, called *footprint* and *footmark* [21, 31, 32, 33, 35].

The footmark measure was motivated by a desire for accuracy: for the first T-1 references of a trace, the working sets contents are truncated windows, potentially underestimating mean working set size. The footprint avoids this by averaging together only the working sets whose T windows fit completely inside the address trace:

$$fp(T) = \frac{1}{N-T+1} \sum_{t=T}^{N} w(t,T)$$

The footprint and mean working set measures do not differ significantly under practical conditions. The definition of mean working set size includes the definition of footprint:

$$s(T) = \frac{1}{N} \sum_{t=1}^{N} w(t,T) = \frac{1}{N} \sum_{t=1}^{T-1} w(t,t) + \frac{N-T+1}{N} fp(T)$$

Since window size is an upper bound for any working set size, $w(t,t) \le t$, and the first sum has an upper bound of $(1+2+3+\cdots+T\text{-}1)/N = T(T\text{-}1)/2N < T^2/2N$. The second term has an upper bound of $fp(T)$. Therefore,

$$s(T) < \frac{T^2}{2N} + fp(T)$$

This reduces to

$$s(T) - fp(T) < 1, \ if\ T \le \sqrt{2N}.$$

In other words, the mean working set size and footprint are within 1 page of each other as long as $T \le \sqrt{2N}$. This will be easily true for programs with good locality, as in Figure 1.

Ding and his colleagues also developed a recursion for $fp(T)$ that would enable calculating footprint for all T in linear time. Here is a derivation of a recursion. Start by writing the footprint for window T+1:

$$fp(T+1) = \frac{1}{N-T} \sum_{t=T+1}^{N} w(t,T+1) = \frac{1}{N-T} \left(\sum_{t=T}^{N} w(t,T+1) - w(T,T+1) \right)$$

The terms involving $w(t,T+1)$ can be reduced to terms involving $w(t,T)$ using the working set size recursion developed earlier. Recall that, when T is increased to T+1, all the runs ending in a reuse interval $>T$ are lengthened by 1. Because every run begins with a page miss, and all the first references in the interval $[1,\ldots,T\text{-}1]$ start runs that extend into $[T,\ldots,N]$, the number of 1s added to the page reference map in the interval $[T,\ldots,N]$ is $mc(T)$. In other words, the same recursion applies to the extended sum, with the same end corrections as before:

$$\sum_{t=T}^{N} w(t,T+1) = \sum_{t=T}^{N} w(t,T) + mc(T) - e(T)$$

As above earlier, $e(T)$ is the number of last references occurring the last T time units. Define $f(T)$ as the number of first references in the first T time units, and notice that $f(T) = w(T,T+1)$. The footprint formula becomes

$$fp(T+1) = \frac{1}{N-T} \left(\sum_{t=T}^{N} w(t,T) + mc(T) - e(T) - f(T) \right)$$

Applying the definition of *fp*(T) and recalling that $m(T) = mc(T)/N$, this becomes the desired recursion,

$$fp(T+1) = \frac{N-T+1}{N-T} fp(T) + \frac{N}{N-T} m(T) - \frac{e(T)+f(T)}{N-T}$$

This messy expression can be computed in linear time, because *mc*(T), *e*(T), and *f*(T) are all computable in linear time.

Chen Ding was not happy with the messiness of algorithms for computing footprint. He and his students defined a new measure they called footmark. Footmark would be like WS but would contain additional terms that compensated for the short working set windows during the initial segment of the trace. They divided an address trace into three regions:

- The initial segment of length *T*-1. In this interval the working set sizes have the form *w*(*t,t*), effectively a window smaller than *T*.
- The second segment has length *N*-*T*+1. In this segment the working set sizes are of the form *w*(*t,T*). This segment includes all the windows of length *T* as in footmark.
- The third segment is the last *T*-1 references of the trace. In this segment a series of phantom working sets of sizes *w*(*N,k*) are defined with progressively shorter windows *k* looking back from the end of the trace. They are "phantoms" because they are not actually observed in a real cache. The idea is *w*(*N,k*) can be paired with *w*(*T-k,T-k*) in the initial segment; every pair spans a full window length *T*. After the pairing, the short-window working sets of the initial segment are replaced by full-window working sets. The net effect is that *N* working sets with window *T* define footmark.

These ideas produced the following definition of footmark space-time *FM*(T):

$$FM(T) = \sum_{t=1}^{T-1} w(t,t) + \sum_{t=T}^{N} w(t,T) + \sum_{k=1}^{T-1} w(N,\ k)$$

The pairing argument says that the first and third sums can be replaced with a single sum of *T*-1 terms *w*(*t,t*)+*w*(*N,t*), so that every working set in *FM*(T) has window *T*. Therefore, the footmark is *fm*(T) = *FM*(T)/*N*.

Because the first two sums are the definition of working set space-time,

$$FM(T) = st(T) + \sum_{k=1}^{T-1} w(N,k)$$

We can now apply the working set recursion to define a footmark recursion:

$$\begin{aligned} FM(T+1) &= st(T+1) + \sum_{k=1}^{T} w(N,k) \\ &= s(T) + mc(T) - e(T) + \sum_{k=1}^{T-1} w(N,k) + w(N,T), \end{aligned}$$

where *e*(T) is the number of end intervals ≤*T*. Now consider the working set *w*(*N,T*). A page is visible in that working set if and only if its final reference occurs before time *N*-*T*+1. Thus the contents of that working are precisely the pages whose end intervals are ≤*T*, which is the definition of *e*(T). This cancels the *e*(T) term. The *s*(T) and sum terms combine into the definition of FM(T). Thus

$$FM(T+1) = FM(T) + mc(T).$$

When all terms are divided by N, we get the footmark recursion:

$$fm(T+1) = fm(T) + m(T)$$

where $fm(T)$ is the footmark for window size T and $m(T)$ is the working set miss rate. The initial conditions are $fm(0) = 0$ and $m(0) = 1$.

ACKNOWLEDGMENTS

I am deeply grateful to Jeff Buzen, Chen Ding, Erol Gelenbe, Roland Ibbett, and Adrian McMenamin for many conversations that sharpened the ideas written here. And I fondly remember my early teachers and collaborators in this work, Les Belady, Fernando Corbato, Jack Dennis, Roger Needham, Brian Randell, Jerry Saltzer, Stuart Schwartz, Donald Slutz, and Maurice Wilkes.

REFERENCES

[1] A. Aho, P. J. Denning, and J. Ullman. 1971. Principles of optimal page replacement. *J. ACM* 18, 1 (1971), 80–93.
[2] L. A. Belady. 1966. A study of replacement algorithms for virtual storage computers. *IBM Syst. J.* 5, 2 (1966), 78–101.
[3] L. A. Belady, R. A. Nelson, and G. S. Schedler. 1969. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM* 12, 6 (1969), 349–353.
[4] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. 2015. Optimal cache partition sharing. In *Proceedings of the 44th IEEE International Conference on Parallel Processing*. 749–758.
[5] J. P. Buzen. 1976. Fundamental operational laws of computer system performance. *Acta Inf.* 7, 2 (1976), 167–182.
[6] J. P. Buzen. 2015. *Rethinking Randomness.*
[7] R. W. Carr and J. Hennessy. 1981. WSCLOCK—A simple and effective algorithm for virtual memory management. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP'81).*
[8] P. J. Denning. 1968. The working set model for program behavior. *Commun. ACM* 11, 5 (1968), 323–333.
[9] P. J. Denning. 1968. Thrashing: its causes and prevention. In *Proceedings of the Fall Joint Computer Conference (FJCC'68)*, part I.
[10] P. J. Denning and S. C. Schwartz. 1972. Properties of the working set model. *Commun. ACM* 15, 3 (1972), 191–198.
[11] P. J. Denning and D. L. Slutz. 1978. Generalized working sets for segment reference strings. *Commun. ACM* 21, 9 (1978), 750–759.
[12] P. J. Denning and J. P. Buzen. 1978. The operational analysis of queueing network models. *ACM Comput. Surv.* 10, 3 (1978), 225–261.
[13] P. J. Denning. 1980. Working sets past and present. *IEEE Trans. Softw. Eng.* SE-6, 1 (January 1980), 64–84.
[14] P. J. Denning. 2016. Fifty years of operating systems. *Commun. ACM* 59, 3 (2016), 30–32.
[15] J. FotheringhamJ. 1961. Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store. *Commun. ACM* 4, 10 (1961), 435–436.
[16] G. S. Graham. 1976. *A Study of Program and Memory Policy Behavior.* Ph.D. dissertation, Department of Computer Science, Purdue University.
[17] J. Gray and F. Putzolu. 1985. *The 5 Minute Rule for Trading Memory for Disk Accesses.* Tandem Corporation Technical Report 86.1.
[18] K. C. Kahn. 1976. *Program Behavior and Load Dependent System Performance.* Ph.D. dissertation, Department of Computer Science, Purdue University.
[19] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. 1962. One-level storage system. *IRE Trans. Electr. Comput.* EC-11, 4 (April), 223–235.
[20] P. Li, C. Pronovost, W. Wilson, B. Tait, J. Zhou, C. Ding, C., and J. Criswell. 2019. Beating OPT with statistical clairvoyance and variable size caching. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 243–256.
[21] Y. Yuan, C. Ding, W. Smith, P. J. Denning, and Y. Zhang. 2019. A relational theory of locality. *ACM Trans. Arch. Code Optim.* 16, 3 (2019), 1–26.
[22] A. W. Madison and A. P. Batson. 1976. Characteristics of program localities. *Commun. ACM* 19, 5 (1976), 285–294.
[23] R. J. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Syst. J.* 9, 2 (1970), 78–117.
[24] A. McMenamin. 2011. *Applying Working Set Heuristics to the Linux Kernel.* Master's thesis, Birkbeck College, University of London.
[25] B. G. Prieve and R. S. Fabry. 1976. VMIN—An optimal variable-space page replacement algorithm. *Commun. ACM* 19, 5 (1976), 295–297.

[26] B. Randell and C. J. Kuehner. 1968. Dynamic storage allocation systems. *Commun. ACM* 11, 5 (1968), 297–306.

[27] D. Sayre. 1969. Is automatic folding of programs efficient enough to displace manual? *Commun. ACM* 13, 12 (1969), 656–660.

[28] J. R. Spirn and P. J. Denning. 1972. Experiments with program locality. In *Proceedings of the American Federation of Information Processing Societies Conference 41 (AFIPS)*. AFIPS Press.

[29] X. Wang et al. 2015. Optimal footprint symbiosis in shared cache. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 412–422.

[30] J. Wires et al. 2014. Characterizing storage workloads with counter stacks. In *Proceedings of the USENIX 11th Conference on Operating Systems Design and Implementation*, 335–349.

[31] X. Xiaoya, B. Bao, C. Ding, and Y. Gao. 2011. Linear-time modeling of program working set in shared cache. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*, 350–360

[32] X. Xiang, C. Ding, H. Luo, and B. Bao: 2013. HOTL: A higher order theory of locality. In *Proceedings of the Annual Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*, 343–356.

[33] X. Hu, X. Wang, L. Zhou, Y. Luo, Z. Wang, C. Ding, and C. Ye. 2018. Fast miss ratio curve modeling for storage cache. *ACM Trans. Stor.* 14, 2 (2018).

[34] C. Ye, C. Ding, H. Luo, J. Brock, D. Chen, and H. Jin. 2017. Cache exclusivity and sharing: Theory and optimization. *ACM Trans. Arch. Code Optim.* 14, 4 (2017), 1–26.

[35] L. Yuan, W. Smith, S. Fan, Z. Chen, C. Ding, and Y. Zhang. 2018. Footmark: A new formulation for working set statistics. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (October)*, 61–69. Springer.

Received October 2019; accepted May 2020