Locality in computer archtecture

It is a package of three ideas: (1) computational processes pass through a sequence of locality sets and reference only within them, (2) the locality sets can be inferred by applying a distance unction to a program’s address trace observed during a backward window, and (3) memory management Is optimal when it guarantees each program that Its locality sets will be present in high-speed memory. Working set memory management was the first exploitation of this principle; it prevented thrashing while maintaining near optimal system throughput, and eventually It enabled virtual memory systems to be reliable, dependable, and transparent.

Many researchers and system designers rallied around the effort to understand locality and achieve this outcome. The principle expanded well beyond virtual memory systems. Today it addresses computations that adapt to the neighborhoods In which users are situated, ways to Infer those neighborhoods by observing user actions, and optimizing performance for users by being aware of their neighborhoods. It has influenced the design of caches of all sorts, Internet edge servers, spam blocking, search engines, e-commerce systems, email systems, forensics, and context-aware software.

It remains a rich source of Inspirations for contemporary research In architecture, caching, Bayesian inference, forensics, web-based business processes, context-aware software, and network science. 1 . Introduction Locality of reference Is one of the cornerstones of computer science. It was born from efforts to make virtual memory systems work well. Virtual memory was first developed in 1959 on the Atlas system at the University of Manchester. Its superior programming environment doubled or tripled programmer productivity.

But It was flunky, Its performance sensitive to the choice of replacement algorithm and to the ways compilers grouped code on to pages. Worse, when It was coupled with multiprogramming, it was prone to thrashing, the near-complete collapse of system throughput due to heavy paging. The locality principle guided us in designing robust replacement algorithms, compiler code generators, and thrashing-proof systems. It transformed virtual memory from an unpredictable to a robust technology that regulated itself dynamically and optimized throughput without user intervention.

Virtual memory became such an engineering triumph that it faded into the memory with multithreading and multitasking that no one notices. The locality principle found application well beyond virtual memory. Today it directly influences the design of processor caches, disk controller caches, storage hierarchies, network interfaces, database systems, graphics display systems, human-computer interfaces, individual application programs, search engines, Web browsers, edge caches for Web based environments, and computer forensics. Tomorrow it may help us overcome our problems with brittle, unforgiving, unreliable, and unfriendly software.

I will tell the story of this principle, starting with its discovery to solve a multimillion-dollar performance problem, through its evolution as an idea, to its widespread adoption today. My telling is highly personal because locality, and the attending success of ritual memory, was my focus during the first part of my career. 2. Manifestation of a Need (1949-1965) In 1949 the builders of the Atlas computer system at University of Manchester recognized that computing systems would always have storage hierarchies consisting of at least main memory (RAM) and secondary memory (disk, drum).

To simplify management of these hierarchies, they introduced the page as the unit of storage and transfer. Even with this simplification, programmers spent well over half their time planning and programming page transfers, then called overlays. In a move to enable programming productivity to at east double, the Atlas system builders therefore decided to automate the overlaying process. Their “ one-level storage system” (later called virtual memory) was part of the second-generation Atlas operating system in 1959 [Killing]. It simulated a large main memory within a small real one.

The heart of their innovation was the novel concept that addresses named values, not memory locations. The Scup’s addressing hardware translated CPU addresses into memory locations via an beatable page table map (Figure 1). By allowing more addresses than locations, their scheme enabled aerogramme to put all their instructions and data into a single address space. The file containing the address space was on the disk; the operating system copied pages on demand (at page faults) from that file to main memory. When main memory was full, the operating system selected a main memory page to be replaced at the next page fault.

The Atlas system designers had to resolve two performance problems, either one of which could sink the system: translating addresses to locations; and replacing loaded pages. They quickly found a workable solution to the translation robber by storing copies of the most recently used page table entries in a small high speed associative memory, later known as the address cache or the translation lakeside buffer. The replacement problem was a much more difficult conundrum. Because the disk access time was about 10, 000 times slower than the CPU instruction cycle, each page fault added a significant delay to a Job's completion time.

Therefore, minimizing page faults was critical to system performance. Since minimum faults means maximum inter-fault intervals, the ideal page to replace from main memory is he one that will not be used again for the longest time. To accomplish this, the Atlas Figure 1. The architecture of virtual memory. The process running on the CPU has access to an address space identified by a domain number d. A full copy of the the main memory. The page table PET[d] has an entry for every page of domain d.

The entry for a particular page (I) contains a presence bit P indicating whether the page is in main memory or not, a usage bit U indicating whether it has been accessed recently or not, a modified bit M indicating whether it has been written into or not, ND a frame number FAN telling which main memory page frame contains the page. Every address generated by the CPU is decomposed into a page number part (I) and a line number part (x). The memory mapping unit (MUM) translates that address into a memory location as follows. It accesses memory location d+I, which contains the entry of page I in the page table PET[d].

If the page is present (P= l), it generates the memory location by substituting the frame number (f) for the page number (I). If it is not present (P= O), it instead generates a page fault interrupt that signals the operating yester to invoke the page fault handler routine (PH). The MUM also sets the use bit (13= 1) and on write accesses the modified bit (M= l). The PH selects a main memory page to replace, if modified copies it to the disk in its slot of the address space file AS[d], copies page I from the address space file to the empty frame, updates the page table, and signals the CPU to retry the previous instruction.

As it searches for a page to replace, the PH reads and resets usage bits, looking for unused pages. A copy of the most recent translations (from page to frame) is kept in the translation lakeside buffer (TTL), enabling the MUM to bypass the page table lookup most of the time. System contained a " learning algorithm" that hypothesized a loop cycle for each page, measured each page's period, and estimated which page was not needed for the longest time. The learning algorithm was controversial. It performed well on programs with well-defined loops and poorly on many other programs.

The controversy spawned numerous experimental studies well into the sass that sought to determine what replacement rules might work best over the widest possible range of programs. Their results were often contradictory. Eventually it became apparent that the volatility resulted from variations in compiling methods: the way in which a compiler grouped code blocks onto pages strongly affected the program's performance under a given replacement strategy. Meanwhile, in the early sass, the major computer makers were drawn to multiprogramming virtual memory because of its superior programming environment.

RCA, General Electric, Burroughs, and Univac all included virtual memory in their operating systems. Because a bad replacement algorithm could cost a million dollars of lost machine time over the life of a system, hey all paid a great deal of attention to replacement algorithms. Nonetheless, by 1966 these companies were reporting their systems were susceptible to a new, unexplained, catastrophic problem they called thrashing. Thrashing seemed to have nothing to do with the choice of replacement policy. It manifested as a sudden collapse of throughput as the multiprogramming level rose.

A thrashing system spent most of its time resolving page faults and little running the COP]. Thrashing was far more damaging than a poor replacement algorithm. It scared the daylights out of the computer makers. The more conservative IBM did not include virtual memory in its 360 operating system in 1964. Instead, it sponsored at its Watson laboratory one of the most comprehensive experimental systems projects of all time. Led by Bob operating system and used it to study the performance of virtual memory. (The term Mortal memory' appears to have come from this project. By 1966 they had tested every replacement policy that anyone had ever proposed and a few more they invented. Many of their tests involved the use bits built in to page tables (see Figure 1). By periodically scanning and resetting the bits, the replacement algorithm distinguishes recently referenced pages from others. Bellay concluded that policies favoring recently used pages performed better than other policies; LOUR (least recently used) replacement was consistently the best performer among those tested [Bellay]. 3.

Discovery and Propagation of Locality Idea (1966-1980) In 1965, I entered my PhD studies at MIT in Project MAC, which was Just undertaking the development of Multicast. I was fascinated by the problems of dynamically allocating scarce CPU and memory resources among the many processes that would populate future time-sharing systems. I set myself a goal to solve the thrashing problem and define an efficient way to manage memory with variable partitions. Solutions to these problems would be worth millions of dollars in recovered uptime of virtual memory operating systems.

Little did I know that I would have to devise and validate a theory of program behavior to accomplish this. I learned about the controversies over the viability of virtual memory and was baffled by the contradictory conclusions among the experimental studies. All these studies examined individual programs assigned to a fixed memory partition managed by a replacement algorithm. They shed no light n the dynamic partitions used in multiprogramming virtual memory systems.

They offered no notion of a dynamic, intrinsic memory demand that would tell which pages of the program were essential and which were replaceable something simple like, “ this process needs p pages at time t. ” Such a notion was incompatible with the fixed- space policies everyone was studying. I began to speak of a process’s intrinsic memory demand as its “ working set”. The idea was that paging would be acceptable if the system could guarantee that the working set was loaded. I combed the experimental studies looking for clues on how to measure a program’s working set.

All I could find were data on lifetime curves (mean time between page faults as a function of average memory space allocated to a program). These data suggested that the mean working set size would be significantly smaller than the full program size (Figure 2). In an “ Aha! ” moment in the waning days of

1966, inspired by Belays observations, I hit on the idea of defining a process’s working set as the set of pages used during a fixed-length sampling window in the immediate past. A working set could be measured by periodically reading and resetting the use bits in a page table.

The window had to be Figure 2. A program’s lifetime curve plots the mean time between page faults in a virtual memory system with a given replacement policy, as a function of the amount of space allocated to it by the system. It has an S-shape. The knee, defined as the point at which a line emanating from the origin is tangent to the curve, is the point of diminishing returns for increased memory allocation. The knee memory size is replacement policy can often do quite well with a relatively small memory allocation.

A further significance of the knee is that it maximizes the ratio L(x)/x for all points on the curve. The knee is therefore the most desirable target for space allocation: it maximizes the mean time between faults per unit of space. In the virtual time of the process time as measured by the number of memory references made so that the measurement would not be distorted by interruptions. This led to the now-familiar notation: the working set W(t, T) is the set of pages referenced in the virtual time interval of length T preceding time t [Deeding AAA].

By spring 1967, I had an explanation for thrashing [Deeding ebb]. Thrashing was the collapse of system throughput triggered by making the multiprogramming level too high. It was counterintuitive because we were used to systems that would saturate under heavy load, not shut down

(Figure 3). When memory was filled with working sets, any further increment in the multiprogramming level would simultaneously push all loaded programs into a regime of working set insufficiency, where they paged excessively and could not use the CPU efficiently Figure 3.

A computer system’s throughput Bobs completed per second) increases with multiprogramming level up to a point. Then it decreases rapidly to throughput so low that the system appears to have shut down. Because everyone was used to systems hat gradually approach saturation with increasing load, the throughput collapse was unexpected. The thrashing state was “ sticky’ we had to reduce the ML somewhat below the trigger point to get the system to reset. No one knew how to predict the optimal ML or to find it without falling into thrashing. (Figure 4).

I proposed a feedback control mechanism that would limit the multiprogramming level by refusing to activate any program whose working set would not fit within the free space of main memory. When memory was full, the operating system would defer programs requesting activation into a holding queue. Thrashing would be impossible with a working set policy (Figure 5). The working set idea was based on an implicit assumption that the pages seen in the backward window were highly likely to be used again in the immediate future. Was this assumption Justified?

In discussions with Jack Dennis (MIT) and Less Bellay (MM), I started using the term “ locality’ for the observed tendency of programs to cluster references to small subsets of their pages for extended intervals. We could represent a program’s memory demand as a sequence of locality sets and their holding times: , (Lie, Tit), This seemed natural because we knew that Al , TO), (LA, TO), (LA, TO), programmers planned overlays using diagrams that showed subsets and time phases (Figure 6). But what was strikingly interesting was that programs showed the locality behavior even when it was not explicitly pre-planned.

When measuring actual page use, we repeatedly observed many long phases with relatively small locality sets (Figure 7). Each program had its own distinctive pattern, like a epicenter. We saw two reasons that this would happen: (1) temporal clustering due to looping and executing within modules with private data, and (2) spatial clustering due to related values hose reasons seemed related to the human practice of “ divide and conquer” breaking a large problem into parts and working separately on each. The locality bit maps captured someone’s problem-solving method in action.

These underlying phenomena gave us confidence to claim that programs have natural sequences of locality sets. The working set sequence is a measurable approximation of a program’s intrinsic locality sequence. Figure 4. The first rigorous explanation of thrashing argued from efficiency. The efficiency of a program is the ratio of its CPU execution time to its real time. Real time is longer because of page-fault delays. Denote a program’s execution time by E, the page fault rate by m, and the delay for one page fault by D; then the efficiency is E/(E +med) = 1/(1+MD).

For typical values of D 10, 000 memory cycle times or longer the efficiency drops very rapidly for a small increase of m above O. In a memory filled with working sets (high efficiency), loading one more program can squeeze all the others, pushing everyone into working set insufficiency, collapsing efficiency. Figure 5. A feedback control system can stabilize the multiprogramming level and revert thrashing. The amount of free space is monitored and fed back to the scheduler. The scheduler activates the next waiting program whenever the free space is sufficient for its working set.

With such a control, we expected that the multiprogramming level would rise to the optimal level and stabilize there. Figure 6. Locality sequence behavior diagrammed by programmer during overlay planning. Figure 7. Locality sequence behavior observed by sampling use bits during program execution. Programs exhibit phases and localities naturally, even when overlays are not pre-planned. As we developed and refined our understanding of locality during the sass, I continued to work with many others to refine the locality idea and turn it into a behavioral theory of computational processes interacting with storage systems.

By 1980 we articulated the principle as a package of three ideas: (1) computational processes pass through a sequence of locality sets and reference only within them, (2) the locality sets can be inferred by applying a distance function to a program’s address trace observed during a backward window, and (3) memory management is optimal when it guarantees each program that its locality sets will be present in high- speed memory [Deeding 80]. A distance function D(x, t) measures the distance from a processor to an object x at time t.

Distances could be temporal, measuring the time since prior reference or access time within a network; spatial, measuring hops in a network or address separation in a sequence; or cost, measuring any indiscretions accumulation of cost since prior reference. We said that object x is in the locality set at time t if the distance is less than a threshold: D(x, t) T. The storage system would examine throughput by caching locality sets close to the processor. By 1975, the performance of computing systems, and for predicting throughput, response time, and system capacity.

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

In the simplest queuing model of a virtual memory system, there is a server representing the CPU and a server representing the paging disk. A Job cycles between the CPU and the disk in the pattern (COP], Disk)* CPU meaning a series of CPU-Disk cycles followed by a CPU interval before completing. The number of CPU-Disk cycles is the number of page faults generated by the system’s replacement policy for the mean memory space allocated to Jobs. Queuing network theory told us that every server poses a potential bottleneck that imposes an upper limit on the system throughput; the actual bottleneck is the server with the smallest limit.

We discovered that the well-known thrashing curve (Figure 3) is actually the system doing the best it can as the paging- disk bottleneck worsens with increasing load (Figure 8.) Figure 8. System throughput is constrained by both CPU and disk capacity. The CPU imposes a throughput limit of I/R, where R is the average running time of programs. The disk imposes a throughput limit of I/SF, where S is the mean time to do a page WAP and F is the total number of page faults in a Job. Thrashing is caused by precipitous drop of disk capacity as increased load squeezes space and forces more paging.

The crossing point occurs when R= SF; since F= R/L (lifetime, L), the crossing is at L= S, I. E. , when the mean time between faults equals the disk service time of a fault. Thus a control criterion is to allow N to increase until L decreases to S. Unfortunately, this was not very precise; we found experimentally that many systems were already in thrashing when L= S. Moreover, the memory size at which L= S may bear no relation to the highly desirable lifetime knee (Figure 2). Once we saw that thrashing is a bottleneck problem, we studied whether we could use bottleneck parameters as criteria for load controls that prevented thrashing.

One such criterion was called “ L= S” because it involved monitoring the mean lifetime L between page faults and adjusting load to keep that value near the paging disk service time S (Figure 8). This criterion was not very reliable: in some experiments, the system would already be thrashing when L= S. We found that a “ knee criterion” in which the system adjusted load to keep the observed lifetime near the knee lifetime (Figure 2) was consistently ore reliable, even though knee lifetime was not close to S. Unfortunately, it is not possible to know the knee lifetime without running the program to completion. Y a Job is minimum. The memory allocation that does this is near the knee (Figure 2). Our experimental studies of working-set windows near the knee of its lifetime curve yielded two useful results. One is that a program’s space-time is likely to be flat (near minimum) for a broad range of window sizes. The picture shows how we defined a “ 10% confidence interval” of window sizes. Our theory told us that system throughput would be maximum when space-time for ACH Job is minimum, confirming our claim that a knee criterion would optimize throughput.

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

Figure 10. On comparing the 10% confidence intervals, we found that there was very often a global value of T that would put all programs within 10% of their space-time minima. The average deviation from minimum for this value of T was closer to 5%. The conclusion was that systems with a properly adjusted, single global T value would achieve a working-set throughput within 5-10% of optimal. The final question was: is there another policy that would deliver a lower space-time per Job and therefore a higher optimum throughput?

Obviously, the VEIN (variable pace minimum [Upriver]) would do the Job; but it requires Lockheed. We discovered that the working set policy has exactly the same page-fault sequence as VEIN. Therefore the difference of space-time between WAS and VEIN is completely explained by working-set " overshooting" in its estimates of locality at the transitions between program phases. Indeed, VEIN unloads pages toward the ends of their phases after it sees they will not be referenced in the next phase. Working set cannot tell this until time T after the last reference.

Experiments by Alan Smith to clip off these overshoots showed only a minor gain [Smith]. We concluded that it would be unlikely that anyone would find a non-Lockheed policy that was noticeably better than working set. Thus, by 1976, our theory was validated. It demonstrated our original postulate: that working set memory management would prevent thrashing and would allow system throughput to be close to its optimum. The problem of thrashing, which originally motivated the working set theory, has occurred in other contexts as well as storage management.

It can happen in any system where contention for a shared resource can overwhelm the processes' abilities to move forward. It was observed in the various contenders could overwhelm the shared spectrum by retransmitting packets when they discovered their transmissions being inadvertently Jammed by other transmitters [Abramson]. A similar problem occurred in the Ethernet, where it was solved by the " back-off' protocol that makes a transmitter wait a random time before retrying a transmission [Metcalf]. A similar problem occurred in database systems with the two-phase commit protocol [Thomas].

Under this protocol, transactions try to collect locks on the records they will update; but if they find any scored already locked, they release all their locks and try again. When too many transactions try to collect locks at the same time, they spend most of their time gathering and releasing locks. Although it is not critical to the theory and conclusions above, it is worth noting that the working-set analysis applies even when processes share pages. Among its design objectives, Multicast supported multiprocessor (multithreading) computations. The notions of locality and working sets had to apply in this environment.

The obvious approach was to define a computation’s working set s the union of its constituent process working sets. This approach did not work well in the standard paging system architecture (Figure 1) because the use bits that had to be Rod together were in different page tables and a high overhead would be needed to locate them. Fortunately, the idea of capability-based addressing, a locality- enhancing architecture articulated by colleagues Dennis and Van Horn in 1966 [Dennis], offered a solution (Figure 11). Working sets could be measured from the use bits of the set of object descriptors.

The two-level mapping inherent in capability addressing is a principle in its own right. It solved a host of sharing problems in virtual memories of multiprocessor operating systems [Fairy]. It stimulated a line of extraordinarily fault tolerant commercial systems known as “ capability machines” in the sass [Wilkes 72, 79]. The architecture was adopted into the run time environments of object oriented programming systems. The principle was applied to solving the problem of sharing objects in the Internet

[Kahn]. Thus the situations in which working sets and localities of multithreading and distributed computations apply are ubiquitous today.

Table 1 summarizes milestones in the development of he locality idea. Figure 1 1 . Two-level mapping enables sharing of objects without prior arrangements among the users. It begins with assigning a unique (over all time) identifying handle h to an object; objects can be of any size. Object h has a single descriptor specifying its status in memory: present (P = O or 1), used (U = O or 1), base address (B, defined only when P= l), and length (L). The descriptors of all known objects are stored in a descriptor table EDT, a hash table with the handle as a lookup key.

When present, the object is assigned a block of contiguous addresses in main memory. Each computational process operates in its own memory domain (such as del or do), which is specified by an object table (TO), an adaptation of the page table (Figure 1). The object table, indexed by an object number (such as I or J), retrieves an object’s access code (such as raw) and handle. The memory mapping unit takes an address (I, x), looks up the descriptor for the handle in the descriptor table; finally it forms the actual memory address b+x provided that x does not exceed the object’s size a.

Any number of processes can share h, simply by adding entries pointing to h as invention in their object tables. Those processes can use any local name (I or J) they desire. If the system needs to relocate the object in memory, it can do so by updating the descriptor (in the descriptor table). All processes will get the correct mapping information immediately. Working sets can be measured from the use bits (U) in the descriptor table. 4. Adoption of Locality Principle (1967-present) The locality principle was adopted as an idea almost immediately by operating systems, database, and hardware architects.