Locality of Reference, Patterns in Program Behavior, Memory Management, and Memory Hierarchies

—

VERY rough, partial, incompletely-baked draft.
DO NOT REDISTRIBUTE

(Comments welcome.)

Paul R. Wilson

Abstract

Locality of reference is crucial to the performance of modern computers, but is actually poorly understood. In this paper, we survey issues in locality and memory hierarchy design, attempting to bring together what is known, correct common misconceptions, and clarify what is not known.

We present a unified approach to locality, based on the concept of *timescale relativity*, which simply says that some patterns in program behavior are relevant to issues of caching, and others are not, and that the difference depends crucially on the timescale relevant to a particular cache.

Memory hierarchies use a kind of *online*, *adaptive* algorithm to control caching; such algorithms cannot be studied properly without some understanding of the regularities in the "data" (program behavior) they must process.

We attempt a vertical unification, showing that locality of reference results from regularities in the structure of programs, and from regularities in how memory allocators map program objects onto virtual address space.

Contents

1 Overview

This paper discusses several interrelated issues in the study of locality and algorithms:

- *locality of reference* in programs and its effect implications for memory hierarchy design,
- *principled design* of memory hierarchies,
- *locality properties of data sets more generally* (i.e., not just locality of reference, but regularities more generally) and their importance for the algorithmic analysis of common data structures such as binary trees, and
- *locality effects of algorithm and data structure choices*, bringing together the issues of algorithms' responses to regularities in data and the resulting effects on locality of reference.

1.1 Motivation

1.2 Problems in Memory Management Research

We believe that the study of locality of reference has been hampered by a lack of a clear qualitative conceptual framework and a corresponding absence of a taxonomy for understanding program behavior as it affects memory hierarchies. Similarly, some algorithmic analyses are unduly limited by simplistic assumptions about the regularities—or, more often, the lack of them—in input data sets. We believe that by clarifying some of the basic issues in the analysis of algorithms, we can make it easier to conduct more meaningful empirical study of algorithms, and also provide a useful framework for more realistic formal analysis.

One of our guiding hypotheses is that many kinds of programs and data exhibit at least one of a small set of common regularities, and that progress in the design and analysis of algorithms depends on categorizing those regularities, and relativizing analyses according to basic categories of input data. Conversely, failure to recognize the presence of common regularities in data prevents the development of more refined analyses and valid empirical methodology.

Too frequently, algorithms whose performance is dependent on regularities in their inputs are regarded as too difficult to analyze formally, and only their simplest properties are proved. Typically, results are given for worst-case, best-case, and "average" or "expected-case" performance. (Significantly "expected" performance is usually for unpatterned or simply-patterned random inputs, which are often quite unrealistic.) Beyond that, empirical evaluations are used to assess their performance on real data.

Unfortunately, the empirical evaluations are often quite limited—and difficult to draw general conclusions from—because variations in performance are not properly attributed to significant interactions between algorithms and the data they process. Worse, many "empirical" evaluations embody crucial simplifying the assumptions that systematically bias the results, especially through the use of pseudo-random synthetic input data, and these biases often go unrecognized.[1]

We believe that the situation with many kinds of data-dependent algorithms is less grim than it often appears: many kinds of data exhibit similar gross regularities which can be exploited reliably, and the search for relevant regularities can be constrained by analyzing the algorithms' own sensitivities to common regularities—or insensitivities to other possible regularities. Thus it should be possible to derive a fairly simple and useful taxonomy that allows characterization of the purposes to which an algorithm may be put, and its appropriateness for the purpose. Likewise, recognizing which regularities are relevant allows analysis of data sets, to determine which algorithms may be appropriate for different kinds of data in practice.

1.3 Who Should Read this Paper, and How

[NOTE to the readers of this (early) draft:

Hi friends. Welcome to my brain dump on locality, which some of you have asked to see.

I'm not sure exactly what form this material will eventually take.

I hope to have a complete draft of most of it within a month or two, and package it up as a technical report. By that time, it'll probably be a little less casual (and in places, a little less blunt).

Parts of it will be incorporated into the book I'm writing on memory management, which will cover allocators, garbage collectors, and persistent object stores, as well as memory hierarchies. That book will be a combination of an advanced textbook and a research monograph.

Some of the later sections may not get written soon, or may be pulled out into separate papers, for example the part about algorithmic analysis, which is not written at all and needs some experimental work done.

The part about architectural issues may be abbreviated, but part of it is already written in the form of course notes that I need to structure and clean up. Likewise, I've got some fairly detailed notes on analytic models, so that part should be fairly easy to write.

A Note about citations: this draft is very short on specific citations, but that will get fixed. The final version will probably have 120-200 citations, like my other big surveys. This version has a lot of citations to my own work, partly because I can do them off the top of my head, and partly because those papers have lots of citations of others' work related to this paper.]

1.4 Structure of the Paper

The rest of the paper is structured as follows.

Section 2 discusses basic issues in locality of reference caching policies. (While the overt content is primarily about locality of reference, the same basic ideas can be generalized to apply to many kinds of data-dependent algorithms, including tree algorithms, compression algorithms, result caching, code caching in on-the-fly compilers, etc.) A key concept, *time-scale relativity* is introduced, to help explain when events are relevant to the performance of a system, and which events are likely to make good predictors of those events.

Section 3 discusses prefetching and clustering, **blah blah...**

Section 4 discusses clustering of objects to improve spatial locality... **blah blah**

Section 5 explains issues specific to different kinds of memory systems: high-speed hardware caches, virtual memory, file systems, etc. This section may be skimmed by readers not interested in the details, but may serve as a helpful reference for understanding later sections.

Section 6 attempts to explain deep issues in locality of reference—where it comes from, how locality in programs can be made visible to the memory hierarchy,

[1] For a striking example, see [WJNB95]; use of synthetic traces has been common in the evaluation of dynamic storage allocation algorithms, with little or no validation—introducing strong biases toward some kinds of allocators and away from others.

etc. We explore effects of overall system structure, programming style, and data structure and algorithm choice.

Section 7 attempts to generalize the notions of locality to include regularities in data other than memory reference streams. These ideas may be helpful in analyzing a variety of algorithms whose performance depends on the patterns in input data, such as adaptive tree algorithms, compression algorithms, etc.

Section ?? **[relates data regularities to program regularities, program regularities to object-referencing regularities, and object-referencing regularigies to memory-referencing regularities... trying to bring it all together.]**

Section ?? discusses techniques for the modeling and simulation of memory hierarchies. A major point in this section is that simple mathematical models are often only narrowly applicable, or unsound. Markov models, in particular, are extremely weak and often mis-applied. We also discuss sound techniques, including detailed tracing and simulation.

Section ?? [**presents conclusions**]

2 Basic Locality of Reference

In this section, we discuss locality of reference in moderate depth. We begin with the traditional notions of temporal and spatial locality, which have been the main ideas of locality for several decades. We then introduce the concept of *timescale relativity*, which we believe is essential for understanding policy choices, and use it to explain the relative performance of various policies. We then proceed to refine the simple notions of temporal and spatial locality to include more subtle characteristics of access patterns, and suggest their implications for memory hierarchy policies.

2.1 Memory Hierarchies

[blah blah... define hierarchy, levels, convention that up means smaller/faster and down means larger/cheaper.

Almost all existing memory hierarchies are based on caching recently-used pages or blocks of data in fast memory. Memory is composed of several levels of increasingly fast and increasingly expensive kinds of memory. Because of their high cost per unit of storage, the faster ("higher") levels are generally smaller and the slower ("lower") levels are much larger.[2]

[2] Some authors reverse this sense of "higher" and "lower."

A simple memory hierarchy might include three levels: disk storage used for virtual memory paging space, main memory to hold recently-used pages in dynamic RAM (DRAM) ("main memory" or "core"), and high-speed cache memory to hold recently-used blocks in still faster static RAM (SRAM) ("high-speed cache memory"). High-performance machines typically add one or two more levels of still faster static RAM on the CPU chip itself ("on-chip cache").

For a typical workstation, the paging area is likely to be scores of megabytes of disk space, divided up into fixed-size pages of about 4 kilobytes. Main memory (DRAM) is used to cache pages that have been touched recently; the rest are left in the paging area on disk until they are touched again. The first-level high-speed cache is likely to be about a half a megabyte, divided into blocks of about 32 bytes. (Page and block sizes are generally a power of two in size, e.g., 4096- or 8192-byte pages, and 16-, 32- or 64- byte cache blocks.)

We will use the term "block" generically, to refer to either virtual memory pages or cache memory blocks.

One fact that is important to notice is that each level of the memory hierarchy typically has many hundreds or even thousands of block frames. For example, a 32-megabyte main memory can hold 8,000 pages of 4 kilobytes each, a 512-kilobyte cache memory can hold sixteen thousand blocks of 32-byte blocks each—and even even a 32-kilobyte on-chip CPU cache can hold a thousand 32-byte blocks. As we will see later, this will have important consequences for replacement, prefetching, and clustering policies.

2.2 Temporal and Spatial Locality

The most common fetch policy is to fetch blocks as needed from slower storage; when a page that is not resident in fast memory is touched by a program (e.g., referenced by a load or store instruction), the program is halted briefly and the data are loaded from slower memory into fast memory. This is called *demand fetching*; the point at which the program attempts to access a nonresident page (and implicitly "demands" that it be fetched) is called a *miss* or a *demand fault.*

For virtual memory, this is typically done by trapping to software and having the software perform the disk I/O. (This is called "page fault handling." A small amount of dedicated hardware detects references to nonresident pages and forces a trap to software that implements the caching policy.) Misses are handled by dedicated hardware for most high-speed cache memories, because cache misses are much more common.

When a miss occurs, space must be reserved in fast memory for the demanded block. The address space is conceptually divided into fixed-sized blocks of data. The actual storage for a block in high-speed memory is called a *frame*. (For virtual memory caching, pages of main memory are called *page frames*; we will call blocks of high-speed cache memory *block frames* for consistency, though they are traditionally called "cache lines.")

The frequency of misses is one of the most important measures of how well a memory hierarchy is working. The *miss rate* is the percentage of memory references that cause demand faults.

For a virtual memory, the miss rate is typically extremely low—very roughly one in a million references is a miss (page fault), and the rest are accesses to pages resident in main memory. Virtual memory miss rates *must* be very low for a system to have reaonsable performance; disk accesses are far, far slower than main memory accesses—about *five orders of magnitude* slower. A typical disk takes at least several milliseconds to respond to a request for a page, so it can only respond to only about 100 requests per second, perhaps 200 for a very fast disk. On the other hand, a fast CPU can execute 100 million instructions per second or more (up to several hundred) and roughly one in five of them are loads or stores. Thus the number of references per second is roughly 50 million or more; if just one in a million of these is a miss, the system will spend about half its time executing instructions between faults, and half its time waiting on the disk.

If the miss rate is much higher, the machine will spend most of its time waiting on the disk, because it will typically only execute instructions for a short period before faulting and waiting on the disk for several milliseconds. For example, a miss rate of one in 10,000 may seem good, but it is actually quite bad. It corresponds to executing about 50,000 instructions between page faults. At 100 million instructions per second, executing 50,000 instructions only takes about a half a millisecond—and then the program must wait for several milliseconds for the disk. Actual program execution speed will then be an order of magnitude slower than the CPU speed. If this situation is chronic, most computer owners will buy more main memory, to bring the I/O costs down and program speed up, making the machine more "balanced."[3]

The situation for cache memories is rather different. The difference between cache memory speeds and main memory speeds is much smaller (roughly a factor of between 5 and 50, depending on the cache level). For the fastest (e.g., on-chip) caches, there is usually another level of cache that usually services a miss without having to go all the way to main memory. Cache misses are therefore far cheaper than page faults; miss rates are much higher for these small memories, often on the order of 1 in 100 or 1 in 1000 memory references.

2.2.1 Temporal Locality and Replacement Policies

Temporal locality is the most often-discussed locality property: most programs tend to access the same blocks of memory repeatedly over relatively short periods of time, so keeping a block in fast memory for a while is likely to pay off. If the block is touched again soon, it can immediately be accessed quickly, rather than being fetched from slow storage. If a block goes unreferenced (untouched by loads or stores) for a while, it is evicted to slower storage, so that the block frame of fast memory it occupies can be used to cache another block.

When a block is faulted on, it must be put somewhere in fast memory. A block frame is selected, and its contents are overwritten with the values of the faulted-on block. Typically, the block frame already holds values for some other block that had been cached there, and its contents must be saved in slower memory. In the general case, this requires writing the contents of that block back to slower storage. If that block's values have not changed since it was faulted in, however, the same values are still held in slower storage (the block frame contains a "clean" copy of the block in slower storage). In that case, the values in the block frame can simply be overwritten. If the block has been modified—is "dirty"—the up-to-date values must be written back to slower storage before the block frame can be used to cache the faulted-on block.

At a fault, the memory system must decide which block frame to use to hold the demanded block. Since

[3]This is not always true. For example, many database systems are "I/O-bound", spending most of their time reading and writing disk data and very little time actually performing computations over the data.

this usually requires "replacing" one block with another, we refer to the policy for choosing a page to replace a *replacement policy.* (Alternatively, we can refer to this as *eviction* and an *eviction policy.*)

Most common replacement policies are approximations of *LRU*—Least Recently Used—replacement. When a block is faulted on, the block chosen for eviction is the one that has not been touched for the longest time. Thus the contents of a cache with n frames are always the n most recently used blocks.

Actual replacement policies are usually an *approximation* of LRU, not true LRU. This is because true LRU replacement turns out to be expensive to implement, but good approximations can be implemented quite cheaply.

LRU replacement works rather well for a large variety of programs, and is by far the best known replacement policy; it is the standard by which other policies are judged. Later, we will explain why LRU works well most of the time, and suggest ways in which its performance might be bested. We will also explain why LRU has some convenient properties for studying program behavior, even if the intent is to design a rather different policy that works better.

2.2.2 Spatial Locality and Block or Page Sizes

Another crucial locality property is *spatial locality of reference.* Most programs not only tend to touch the *same* words of memory repeatedly, but tend to touch words that are *near* recently-touched words in the address space. This is called "spatial" locality, because the ordering of words in the address space provides a hint as to which words are likely to be touched soon. It would be more accurate to call this "spatiotemporal" locality, because it has a crucial temporal component—a word is likely to be touched if it is *near* (spatially, in the virtual address space) something that has been accessed *recently* (temporally).

Spatial locality is the reason that memory systems typically use pages or blocks, rather than simply transferring individual words of data from one level of the memory hierarchy to another. If we were to transfer individual words to and from disk in a virtual memory system, the system would be unusably slow—each word transferred would take several milliseconds. By transferring (say) 4 kilobytes at a time, it is possible to load data into fast memory much, much more quickly—roughly a thousand times faster.

Spatial locality is a more subtle topic than it may seem at first glance, because it is *not* independent of temporal locality.

One important fact to notice is that spatial locality for one block size often appears as temporal locality for larger block sizes—i.e., touches to nearby blocks for a given block size often become repeated touches to a single larger block if a larger block size is used.

Another important subtlety is that when programs are written in high-level programming languages, they are written in terms of references to fields of language-level data objects—not memory addresses. How the program-level locality is mapped onto memory-level locality depend strongly on the compiler's layout of objects and on the allocator's placement of those objects in memory. This will be discussed in depth in Section ??.

2.3 Where does locality come from?

While locality of reference is a crucial property for computer systems, and a lack of it would effectively bring the computing world to a halt, little is actually known about it. In particular, the sources of locality have never been examined in a systematic and thorough way. By and large, locality is regarded as a mysterious property, which computer architects and operating system designers exploit.

There are a few exceptions to this generalization, of course; in some cases, the locality characteristics of particular kinds of simple and regular algorithms are quite well understood. By and large, however, only simple special cases are understood very well at all.

In the following, we'll sketch a simple conceptual framework for understanding what locality is and why it exists.

The first question about locality of reference should be *what is locality?*. We've given a couple of simple examples of locality properties, namely simple temporal locality and simple temporal locality. In our terminology, these are just two of many kinds of locality, which is a necessarily vague concept. Locality is a very general term that means something like "regularities in program behavior," especially exploitable regularities, and especially regularities that are exploitable for caching purposes.

2.3.1 Multiple Levels

The second question about locality of reference should be *reference to what?* A normal memory hierarchy exploits program references to *virtual addresses*; spatial locality is a regularity in the referencing of things that are nearby in the virtual address space; the memory hierarchy exploits those regularities in it's caching policy, i.e., in its mapping of virtual storage to physical storage.

Notice that *programmers don't usually write programs at this level of abstraction.* Most programs are written in high-level languages, with references to program objects such as scalars, arrays and records. Thus there is a logical level of program locality—the pattern of a program's references to language-level entities like records—which is somehow mapped onto virtual address space by programming language implementations.

So there are at least two important levels of locality, and the locality exploited by memory hierarchies is a function of at least two things: how programmers write programs, and how language implementations map language-level behavior onto the virtual address space.

There are other, higher levels as well, however. Software is typically structured in layers, and each layer's implementation can affect locality.

Programmers choose language-level representations of conceptual, application-level entities. For example, a programmer might choose to represent a set of people using a list, a tree, or a hash table of records; each of these has very distinctive locality effects at the at the level of references to individual program objects. For sophisticated algorithms and data structures, there may be several levels of mapping between the conceptual objects (and high-level algorithms) and the language-level objects (and low-level algorithms).

2.3.2 Programmers' Problem-Solving Strategies

Divide-and-conquer strategies. Programmers often solve programming problems by dividing them up into subproblems, dividing those up into smaller subproblems, and so on, until very small problems can be solved in actual pieces of code. This hierarchical plan structure often affects memory-referencing patterns, causing frequent references to data objects during certain phases. Sometimes, the hierarchical task structure is reflected in the pattern of repeated references to memory locations, but sometimes it is not. A particular kind of problem-solving phase of a program may always reference the same data structures, or it may reference different data structures, but in the same way.

Loops. Many programs loop to repeat the same action over and over again; this may cause repeated accesses to the same memory locations, or it may cause different memory locations to be accessed in the same way. Loops often have a major effect on locality; loops over a small amount of data may repeatedly touch the same items, resulting in excellent temporal locality. Loops over larger amounts of data have very different effects.

More complex control flow.

2.3.3 Problem Structure and Input Data

Some programs' referencing behavior is heavily dependent on the characteristics of their input data. A compiler, for example, may act very differently when compiling a file full of small functions using simple constructs than it does when compiling a file containing a few very large, complex functions. An interactive word processing program may behave very differently when given different commands.

In these cases, the regularities in the inputs may be a source of regularities in the program's behavior, which is not evident from an examination of the programs. A compiler's behavior may be largely determined by the coding style of its input source programs, and a word-processor's behavior is strongly affected by its users' work habits—e.g., how often they simply type in text, and when they perform commands like searches or reformatting over the entire document.

2.3.4 Compilers' object layout choices

Compilers affect spatial locality by grouping fields of objects together in some order. Most compilers lay out the fields of a record (or class instance) in consecutive words of memory; accesses to multiple fields of the same object are accesses to nearby memory. This grouping is especially important for large arrays **[blah blah...]**

2.3.5 Placement choices by allocators, compilers and linkers

The locality of references to executable code and statically allocated data are strongly affected by compilers and linkers. Most compilers organize machine code into object files in roughly the order procedures are defined in source files, and lay out statically-allocated data in a similar order. Linkers may combine the contents of object modules in a somewhat different order in executable files.

Usually, compilers and linkers approximately preserve the order of definition of variables, and the order of definition of procedures, but may group the variables from multiple files together, separate from the code.[4]

The preservation of definition ordering seems to have relatively good effects on locality, because the definition ordering often reflects the problem-solving strategy used by the program. The hierarchical structure of execution turns out to result in good spatial locality of accesses to the code and variables, because it is often strongly correlated with the plan structure used in problem solving and the phase structure of program execution.

2.3.6 Memory Re-allocation

Memory allocation strategies have a major effect on locality. **[blah blah blah...]**

Stacks. Stack allocation often has excellent locality of reference. An important example of this is the activation stack for a typical program in a conventional language; local variables and control information are allocated in activation records when procedures are entered, and deallocated when the procedure is exited. Typically, the stack height does not vary dramatically over a short period of time—the stack grows and shrinks repeatedly by one or a few activation records, reusing the same area of memory for a very large number of activation records. The active part of the activation stack—the top—may be touched millions of times per second, and stay cached in fast memory.

Stacks created by programs for other purposes often have similarly high temporal locality; if allocated in contiguous memory, as activation stacks usually are, spatial locality is likely to be excellent as well. A single virtual memory page may contain the whole stack, or the active part of a large stack, and that page may be referenced very, very often.

Even if the stack is represented as a linked list in heap memory, temporal locality is very likely to be excellent; spatial locality is likely to be excellent as well, depending on the heap memory allocator used and the pattern of allocation that creates the stack.

Roughly stack-like use of heap memory. Heap memory allocation can have major effects on locality; for many programs, heap data account for most of the memory used, and references to objects on the heap are the dominant locality consideration.

Many programs use heap memory in roughly stack-like ways, at the program level, even when they are not using stack data structures. Many objects are quite short-lived, and freed—that is, their storage is returned to the free memory pool—very shortly after they are allocated. This is often true because most objects are created and used to solve small subproblems (near the leaves of the problem decomposition graph), and then discarded.

An allocator can take advantage of this by reusing recently-freed memory in preference to memory that has been free for a longer time.

Notice that in this case, the program itself may have "bad" locality of reference, in that it touches many different objects over a relatively short period of time, but its actual locality of memory referencing may be excellent; the allocator can map different language-level objects onto the same virtual address ranges over time, so that the same memory is referenced over and over.

Reuse of language-level objects for different conceptual objects. In a similar way, programmers may reuse memory by mapping multiple conceptual objects onto the same language-level object over time.

For example, in FORTRAN or C programs, it is common to use a statically allocated array many times, holding different data each time. Consider a FORTRAN program that repeatedly reads data samples from a file into a 1024-element array and performs an FFT on each set of 1024 samples. Conceptually, each set of samples is a different entity, but the programmer has mapped them onto one language-level array.

More generally, **[blah blah blah...]**

Use of different language-level objects for the same conceptual objects.

[4] Linkers also often group literal data together, and initial values of global variables together, separately from uninitialized variables. This separation allows one copy of code and constants to be shared between multiple processes running the same program.

2.3.7 Pragmatic Factors

Limitations of existing hardware.

Limitations of existing software.

2.4 Some Useful Concepts, Terminology, and Techniques

In this subsection, we will introduce some convenient terminology and techniques for studying memory hierarchies, and for the study of algorithms and sequence behavior more generally.

2.4.1 Localities and Working Sets

The term "locality" was used originally used (in a caching context) to refer to a specific collection of items used by a phase of a program—it was a "count" term (e.g., "*this* locality" vs. "that locality") rather than a mass term (e.g., "*this much* locality" vs "*that much* locality").

The basic idea, and it's still a good one, was that program phases tend to preferentially touch certain items, and the set of item touched by a phase is called a "locality." Notice that a locality is a *temporally related set of items*—things that tend to be accessed at about the same time, not a spatially related set in terms of address space. The items in a locality may be spread across the address space, but still be "neighbors" in time, in terms of the ordering of programs' accesses to data.

More recently, the term "locality" has generally been used in a more flexible way, as we do in this paper, and the old sense has been largely superseded by the term "working set." A working set is the set of items "worked with" by a phase of program execution—a set of items that tends to be touched at about the same time.

There is also a very specific technical sense of the term "working set," defined by Denning. We will discuss this technical sense later in the paper, but for the time being we will use the intuitive sense, in which a working set (or locality) is a set of blocks (or objects) that tend to be accessed "together," during a program phase.

It is important to realize that working sets (or localities) are not necessarily *disjoint* sets—the working sets of two phases may overlap, if both phases reference some of the same items.

Working sets are also timescale-dependent. If we look at program's behavior over short timescales relevant to small caches, we may notice that it tends to have different working sets for different short phases. If we look at the same program over a large timescale, we may notice that there are larger phases with larger working sets, and the working sets of the larger phases include the working sets of the smaller phases.

Working sets may be more or less distinct. A certain kind of program phase may always access the same items. On the other hand, different occurrences of the same kind of phase may access some of the same items, but some different ones too.

2.4.2 Latency and Transfer Time

The cost of fetching a block from slower storage has two components: *latency* and *transfer cost*.

Latency is the "startup time" required to initiate the transfer, e.g., detecting that a block is not present in fast memory, signaling the slower memory device, and whatever that device must do to prepare to transfer the data. Generally, latency is independent of the amount of data to be transferred.

Transfer time is determined the amount of data transmitted and the rate at which data can be transferred once the transfer has been started, e.g., how quickly successive bytes or words of data can be streamed from slower to faster memory. (This rate is usually known as "bandwidth," by analogy to the information rate of a communications frequency band. Bandwidth (transfer rate) is inversely related to transfer costs—the more bandwidth you have, the faster you can transfer data.

In general, it is usually easier to increase bandwidth than to decrease latency. In solid-state components, latency is often determined by physics—the speed of swithching and signal propagation, and ultimately bounded by the sizes of devices and the speed of light. In moving-media memories (e.g., disks), it is quite difficult to make physical parts move extremely quickly due to mechanical problems.[5]

In contrast, increasing bandwidth is often easier, though expensive, through the brute-force approach

[5] This is not to say that latencies do not decrease with improving technology—e.g., increasing levels of integration so that the distance between components on a chip is shorter. It's just that such engineering is fairly difficult, even if cost is no object. For example, a disk head's inertia increases with the square of its velocity, making it difficult to double the speed at which it moves without introducing mechanical problems such as bouncing against the limit of its "throw" and taking longer to "settle."

of adding parallel hardware. By using more wires (or on-chip traces), solid state devices can communicate more bits simply by multiplying increasing the width of the data paths. By using more disks, it is possible to read more data off of disk in a given amount of time.[6]

Latency and transfer costs of magnetic disks. For a magnetic disk, the main contributors to latency are the *seek time* and the *rotational latency*. The seek time is the time it takes to move the magnetic read/write head to the appropriate track of the disk, and the rotational latency is the time spent waiting until the desired block of data comes under the read head. Both of these costs are are actually rather variable—depending on where the read head is positioned relative to the desired track, and the rotational position of the disk when it gets there. The overall access time may vary by a factor of two or three, e.g., from 3 to 8 milliseconds for a fast disk. (For most discussions, however, the access time is assumed to be a constant "average" time and variations are ignored.

Other time costs may also contribute to latency, such as page fault handling (trap handling, plus the cost of the routines that decide what command to issue to the disk, etc.). Due to the extreme difference in speed between moving parts and solid-state switching devices, these costs are usually much less than the cost of the disk seek and rotation.

The transfer rate of a magnetic disk is essentially the rate at which bits can be read off of (or written to) the disk, once the desired bits begin coming under the read/write head. This is determined by the density of bits within tracks, and the rotational speed of the disk. A very fast disk can generally deliver bits at a rate of roughly four megabytes per second, or one four-kilobyte page per millisecond. Slower disks may support half that transfer rate.

Latency and transfer costs of solid-state memories. For solid-state memories, the main contributors to latency include signal delays and setup times within memory modules.

A main (DRAM) memory module is usually internally structured as a logical 2-dimensional array, and an entire row of bits is read from this array into a special buffer. (Once the row has been read into the buffer, successive words from that row can be sent very quickly.) **[blah about static RAM...]**

The transfer times of silicon memories are mostly due to limitations on signaling speeds because of the physical characteristics of the connections. **blah blah blah...**

2.4.3 Balancing Latency and Transfer Time.

An important issue in memory hierarchy design is *balancing* latency and transfer time. In general, there is some tradeoff between them; the exact tradeoff is dependent on spatial locality, but an approximate tradeoff works fairly well for most programs.

By making the block size larger, we can transfer more data at each miss. If spatial locality is good, this will load the needed data into fast memory more rapidly, by reducing the number of misses—and the contribution of latency costs to the overall cost of transferring data. If spatial locality is excellent, then transfer costs will not go up much at all—essentially the same amount of data will be transferred, in fewer and larger units.

On the other hand, if spatial locality is poor, having a too-large block size will increase costs in two ways:

- *Increased transfer times.* If much of the extra data transferred at each miss turns out not to be used before it is evicted, we will unnecessarily increase the transfer time for miss handling.
- *Cache pollution and increased misses.* The fetching of useless data will *pollute* the cache, evicting more useful data. If those data are evicted before being touched again, when otherwise they would not be, then the overall miss rate will increase, indirectly incurring both latency and transfer costs.

In general, the block size should be chosen to trade off between two potential problems: fetching too little data and thus requiring more misses to load data into fast memory, or fetching more data than necessary and perhaps polluting the cache.

This is somewhat oversimplified, however, because we may *prefetch* extra blocks—initiating fetches of several blocks at once—rather than transferring one block

[6] These approaches are not always easy, however. Increasing the width of data paths may introduce other costs, by increasing the amount of power needed and exacerbating electrical isolation and heat dissipation problems, or by increasing fanouts of circuits, which may interfere with making them as fast as possible. Increasing the number of pins used to communicate between chips may seriously decrease the yield of a manufacturing process as well as exacerbating power problems, because pins are one of the greatest sources of failure. At the disk level, increasing the number of disks used puts further demands on the communication channels between disks and memory.

at a time. (In that case, the block size may be smaller than the "fetch size," i.e., the amount of data fetched at each fetch. The block size may be chosen in part with respect to the cost of maintaining the mappings that record which blocks are where in fast memory, and in part to provide flexibility in fetching.)

A *block* is really a unit of *address translation*—we map a block of virtual address space to a block frame in fast memory—which can be decoupled from the *unit of data transfer* by transferring more (or perhaps less) than one block at a time. Prefetching memory systems may transfer more than one block at a time, increasing the *fetch size* without increasing the block size. For the moment, however, we will ignore this complication, which is discussed in depth in Section 3.

In deciding on the block size (or, more generally, the fetch size), spatial locality must be taken into account. The ideal block size for programs with good spatial locality is much higher than for programs with bad spatial locality, because the extra transfer time at each miss is much more likely to pay off. Unfortunately, the block and page sizes are generally determined by the hardware designers, and "reasonable" sizes are chosen; for any given application, this may be a good size, or it may not.

In general, a good choice of fetch size is one where the latency and transfer times are roughly balanced. For example, suppose we choose a block size so that the latency and transfer time are exactly equal. In that case, a program with very good spatial locality will perform more (and smaller) fetches than would be ideal, but the overall cost will *never be more than a factor of two worse than with very large blocks*—the same amount of transfer time will be incurred fetching small blocks as large ones, and the extra latency cost will be less than or equal to that.

Increasing fetch sizes beyond this "breakeven" point will rapidly reach a point of diminishing returns, even for programs with excellent locality.

For example, consider a cache with a transfer time of 16 units and a latency of 16 units, for a total miss cost of 32 units. Doubling the transfer size will increase the transfer time to 32 units; in the best case, excellent spatial locality will ensure that the total transfer time is unchanged—for each two transfers we'd done previously, we'll do one that takes twice as long. In this best case, the overall latency cost will be cut in half by halving the number of misses.

Another doubling will decrease the latency cost by half again, but the overall (remaining) cost by at most a sixth, because the transfer cost is not reduced. After a few doublings, the decreases in latency cost are negligible, because the the overall contribution of latency costs is already small, and the transfer time dominates.

In contrast, the transfer cost per miss increases by a factor of two with each doubling, and for programs with *poor* spatial locality, the consequences may be severe, due to increased miss service times and increases in misses due to cache pollution.

Conversely, decreasing the block size far below the balance point runs similar risks—if spatial locality is good, halving the block size will straightforwardly double the miss rate; a very small block size will only pay off if spatial locality is very poor.

In general, the block size should be chosen to be near the balance point, but perhaps somewhat larger if programs generally exhibit good spatial locality.

Assessing the effects of block size and replacement policy [blah blah... miss rate is not a good figure of merit because it leaves out transfer costs...]

2.4.4 Compulsory and Capacity Misses

Traditionally, misses for an LRU replacement policy are divided into two categories:

- *Capacity* misses, due to the cache being too small to hold blocks between touches to them, i.e., misses to blocks that have been evicted and then touched again, and
- *Compulsory* misses, due to the first touches to blocks.

A miss due to the first touch to a block is called *compulsory* because no cache would be large enough to hold the block—any cache would suffer a miss for the block, independent of replacement policy or size.

A miss at a repeated touch to a block is called a *capacity* miss because for some large cache size, the LRU replacement policy would be able to cache the block between touches and avoid the miss.

Later, **[in the next section? Or move much later?]**, we will refine this categorization of misses. In some situations, compulsory misses are expensive, but in other situations, they may be very cheap. The concept of capacity misses also needs further refinement, especially when different replacement policies are being compared, because different baselines make sense in different situations.

2.4.5 Basic Profiling and Simulation Techniques

Reference Traces and Trace-driven Simulation. Studies of locality and memory hierarchy design are often done in simulation, rather than by building actual memory hierarchies and actually running programs in them. It is much easier to build a simulator than a real computer, and it is much easier to experiment with variations in a software simulator than in actual hardware.

To perform simulations, memory-referencing behavior of real programs can recorded in a *trace file*, as a sequence of *trace records* that record what the program does that is relevant to the memory hierarchy. For most purposes, that is just a record of the references to memory—loads and stores to particular addresses—and perhaps instruction fetches. The contents of the memory locations are not actually relevant, because most memory hierarchies do not adjust their caching to the *contents* of memory, only to the pattern of virtual addresses that is accessed.

Typically, reference traces are gathered by executing programs in simulation, using an interpreter that interprets machine code and simulates an actual CPU. Alternatively, programs may be *instrumented* by modifying their instruction sequences to not only perform the normal computation, but to record the memory-referencing that *would have* been done by the unmodified program.[7]

Once gathered, the reference trace can be processed by a memory hierarchy simulator which simulates the relevant actions of a given memory system when each event is encountered—e.g., moving a page from slow to fast memory, changing the LRU ordering of pages, evicting a page, etc.—and records the events that would affect performance. As an optimization the reference trace may be fed directly into a simulator, performing simulation "on the fly" while a program is being traced. This requires re-tracing the program for each simulation, but avoids the need to actually store the trace in a file. Since reference traces may contain billions of records, this can save considerable storage space.

The LRU distance string. Using an LRU simulator, which maintains a record of the LRU ordering of blocks, a reference trace can be transformed into an *LRU distance string*. Where the original trace indicates the order of touches to particular words (or blocks) of memory, the LRU distance string records the LRU positions of those blocks at each touch.

That is, for each reference in the original string, the distance string records how long it has been since the last touch to that block, in terms of the number of other blocks touched in the meantime. It is therefore a fairly direct reflection of the actual temporal locality of a trace. It more directly reflects the patterns of repeated touches to blocks, independent of which blocks they are touches *to*.

A simple way to transform the reference trace into a distance string is to process the records in the trace sequentially, maintaining a data structure that records the LRU ordering, for example, a sorted linear list or tree. (The list only records the block numbers, not their contents.) Each reference in the trace is processed by searching the ordered list and determining the position of the block in the list; this LRU position is emitted as an item in the distance string. In addition, the just-touched block's record is moved to the end of the list, making it the most-recently-touched item in the ordering and pushing intervening items one position the other end.

We will refer to the list (or other data structure) recording the LRU ordering as an LRU *queue*, because items are inserted at one end of the list, and get pushed along as other items are inserted. This is not a simple queue, however, since items may be searched for, have their position recorded when they are found, and be removed from the queue so that they can be reinserted at the end.

Optimized implementations are possible, of course, for example using a balanced or adaptive binary tree to implement the linear ordering "queue."

[7] The best example of this is Larus' QPT tool, which rewrites an executable program to make a self-tracing version [BL92].

Cmelik and Keppel's [CK93] amazing Shade tool (for Sun SPARC machines) incorporates both interpretive and executable-rewriting features, using a dynamic compiler to speed interpretation by decompiling, annotating and recompiling code on the fly; these rewritten fragments are executed directly to avoid the cost of true interpretation. (A cache of these rewritten code fragments is maintained, so that the costs can be amortized over repeated executions of the same code fragment.)

A different approach to self-tracing code is taken by Wilson and Balayoghan's VMtrace tool, which slightly modifies programs to trace themselves using virtual memory access protection; most pages are kept access protected, and a fault handler records the order in which they are faulted on [?].

Blah blah on limitiations of software tracing, and hardware approaches... BACH, etc.

Note that if we fix the length of the LRU queue, and delete items when they reach the head of the queue, we can easily construct a simple simulator for an LRU replacement policy. The queue ordering represents the mechanism for scheduling blocks for eviction. Pages that go untouched for a long period drift toward the head of the queue, while pages that are touched are moved to the tail, and saved from eviction.

The length of the queue represents the size of fast memory, i.e., the number of block frames, and the queue ordering represents the LRU ordering maintained by the replacement policy; a touch to an item that is not yet in the queue counts as a miss, and a touch to an item that is in the queue counts as a hit.

A slight improvement lets us use this basic implementation strategy to simulate many sizes of memory, not just one, during a single pass through a trace. Notice that the elements of an LRU queue of m blocks are a subset of the elements of an LRU queue of $m+1$ blocks. For example, the five most recently touched pages include the four most recently touched pages, plus one more. A longer LRU queue is always a prefix of all shorter LRU queues, so we can combine them into a single arbitrarily-long queue which represents any number of fixed-lenth queues by combining their tails.

The LRU distance histogram and LRU miss histogram. The LRU distance string can be summarized by the *LRU distance histogram.* This is simply a histogram that records how many times each LRU queue position was touched during the processing of a trace, or during an interesting subinterval within the trace.

The LRU distance histogram can be computed from the LRU distance string, by simply counting the number of hits to each queue position in the entire string. This can also be done on the fly, during LRU processing of the trace, by simply incrementing counters in an array (one element for each queue position) rather than actually emitting the distance string.

The LRU distance histogram is interesting because it shows the locality characteristics of a program in a way that is independent of any particular memory size, but which can be interpreted with respect to any memory size of interest. For any LRU replacement policy, it records *which events are relevant to which sizes of memory.* For an LRU memory of size m, touches to LRU queue positions 1 through m represent hits, and for touches to positions m and above represent misses.

Because of this, we can trivially compute the number of misses for every memory size, by simply adding up the numbers of hits to higher LRU queue postions. **...it's a simulator... blah blah... stack property... ...blah blah... what "typical" distance and miss histograms look like... ...caveats...**

2.5 Some "Typical" Program Behavior

Most programs exhibit some common locality properties, which it will be useful to distinguish. We will introduce some terms, which should easily be understandable from the preceding discussion.

2.5.1 Simple Heat (Hot/Cold Reference Skew)

Hot/cold reference skew is perhaps the simplest kind of locality—a program may touch some items much more often than it touches others. For example, the activation stack of a program may stay within a single virtual memory page, and that page may be touched every few instructions throughout the whole execution of a program. Other pages may be touched much less often.

All other things being equal, it is better to cache "hot" (frequently-touched) blocks than to cache "cold" (infrequently-touched) blocks. In many cases, however, other things are systematically *not* equal, and caching based solely on "heat" (the number of times a block is touched) does not work well.

In real programs, it is common for some blocks to be very hot and others quite cold. Over a run of a program, some blocks may be touched millions of times, and others once or a few times, with still others being some intermediate number of times.

On average, the hot/cold distribution of blocks for programs in general is roughly exponentially decreasing—very few blocks are very hot, and few blocks are warm, and very few many blocks are much cooler.

For any given program, however, the distribution may not be at all smooth. There may be distinct sets of blocks which have similar heat, and no blocks with intermediate heat values. (For example, there might be very hot, hot, and cool blocks, but no blocks with heat values intermediate between those, i.e., no medium-warm or cold blocks.)

Figure ?? shows a heat distribution for the pages during a particular run of a particular program (**[which program?]**). The horizontal axis represents

Figure 1: Heat Distribution

Figure 2: Recency Distribution

address space (block number) and the vertical axis represents the total number of touches to each block during short segments of program execution.

2.5.2 Recency Skew and Recency Distributions

Another common property of real programs is *recency skew*—at any given time, a program is much more likely to touch a block that it has touched *recently* than one that it has not touched for a long time. This is the principle behind LRU caching.

More generally, a program or a program phase may tend to touch things again after touching a certain number of other things. For example, a simple loop over 100 blocks may touch the 100th most recently touched block, but never the 70th or the 121st most recently touched block. This is what the LRU distance histogram illustrates—the recency distribution of a program.

Figure 2 shows the recency distribution for [**same program**], over an entire run. Figure 2

2.5.3 Phase Behavior

The hot/cold skew or recency skew of a program often varies over time. If we look at simple heat histogram or LRU distance histogram for a particular program phase, it may look very different from the histogram for the entire program run.

Figure 3 shows the evolution of block heat for the same program as in Figure 3. Here the horizontal axis is time, increasing toward the right. (We have divided execution into "time slices" of a few [**million**] instructions.) The vertical axis represents adddress space address space (block numbers). (This corresponds to the horizontal axis of Figure 1.) The gray scale represents the number of touches to each block in a particular unit of time, which corresponds to the vertical axis of Figure 1. The gray scale is logarithmic—slight differences in tone represent fairly large differences in heat.

Figure 3: Heat Distribution Over Time

Figure 4: Recency Distribution Over Time

Figure 4 shows the evolution of the recency distribution over time in a similar fashion. (Time increases to the right, and MRU queue position increases toward the top.) Again, the gray scale is logarithmic, so visible patterns represent fairly strong regularities.

[**ramble about patterns in figures a little**]

2.6 Demand Fetching Policies and Timescale Relativity

A replacement policy is implemented by an adaptive algorithm, whose job is to predict which blocks of memory are likely to be needed soon, and which are not. The only information a general-purpose replacement policy can use to make this prediction is the actual string of references up to the point where a fault occurs. This string may contain many complex patterns, and a replacement policy could conceivably exploit *any* information in the string. In designing a replacement policy, the goal essentially to filter the reference string to find highly predictive information—and the *robustly* predictive information. Not only is it desirable to have a policy that makes accurate predictions, but the heuristics should work for a wide variety of programs' access patterns and seldom (or, ideally, never) fail disastrously.

In general, we can view the replacement policy's job as having three parts:

1. detecting regularities in past references to memory,
2. using the detected regularities to predict future references, and
3. using the predictions to determine a good choice of which blocks to cache and which to evict.

[**We're talking about demand fetching policies only, here, no prefetching. The ideas will be generalized later, in the discussion of prefetching and clustering.**]

[**Blah blah... can screw up on any of these... few studies separate out the three issues... don't have a good grasp...**

We'll talk about optimal and LRU before talking about other policies... these are interesting because they provide an interesting pair of baselines... optimal because it's optimal...

LRU because it's common, implementable, and competitive with optimal. Any good policy must resemble LRU to some degree...]

2.6.1 The Holy Grail: Optimal Replacement

The replacement policy's main task is to discriminate between blocks that are worth keeping in memory, so that they can be touched again without a fault, and blocks which should be evicted sooner to make room for "more important" data. The ideal replacement policy is be one that could accurately predict the time of the next touch to each block, and keep in memory those that will be touched soonest. This policy is not implementable in practice, because complete accuracy for arbitrary programs requires being able to see into the future; this is not possible in a real system where the memory hierarchy must respond to program behavior as it happens. (However, it turns out to be easy and useful to implement such an optimal policy in simulations, as will be discussed later.)

Note that the ideal (demand fetching) policy only discriminates between blocks that will be touched soon and those that will not be touched soon, where "soon" is *relative to the memory size*. For a memory of size m, the ideal policy retains the m blocks that will be touched soonest, and evicts the blocks that will only be touched later. Blocks that will only be touched later (after touching m distinct blocks) can be evicted at any time—e.g., at the next page fault. Such a block is doomed not to be touched again until *after* it has been evicted anyway, so its space might as well be made available for caching something else in the meantime.

[explain or fwd ref. explanation of why this is always true]

Thus once the last reference to a block *for a while* occurs, it can safely be evicted immediately to make space. Here, "for a while" means for a period during which m or more other pages are touched.

Note that if a page will be touched sooner than that, *it doesn't matter much how much sooner*—it will be a mistake to evict it. The page will simply be evicted and then faulted in again when touched, so it will cost exactly one fault to make this mistake once.

Note also that if a block will not be touched for m block faults or longer, *it doesn't matter how much longer*. It doesn't matter exactly which of these blocks will be touched in what order. If all of them will not be touched "for a while" that's long enough that they all should be evicted. The blocks can be evicted in any order, as needed to make room for faulted-on blocks.

From this we can see that even a perfect replacement policy doesn't need to make very *precise* predictions with respect to future access patterns, or even the times until the next accesses to blocks: it only needs to *accurately* determine whether a block is in the "touched soon" set or the "not touched soon" set.

If we assume that the prediction function is accurate, the only situation in which precision matters to the effectiveness of the algorithm is *when there is a choice between blocks that will be referenced at roughly the same time*, and *both are touched after roughly m other blocks are touched.*

From this we can conclude several basic things about how to construct a good replacement policy, which approximates the ideal:

- Details of future access patterns don't usually matter much.
- Details of *past* access patterns only matter if they are good predictors of the crucial characteristics of future access patterns.
- The crucial information about access patterns is *timescale relative*, i.e., keyed to the memory size in question, and when the program will have touched that many distinct pages.
- The replacement policy needs only to make a binary distinction for each page—to keep it or evict it. This judgement is made relative to *other pages*, however; it depends on *how many* other pages will be touched sooner (but not which ones).

2.6.2 LRU and Looping Behavior

Most replacement policies are based on a simple kind of heuristic, where touches to blocks are used to predict touches to blocks, and no complex or subtle patterns are recorded or used for prediction. LRU, for example, only records the relative ordering of the *last* touches to the cache-resident blocks. All other information about previous patterns of touches to those pages is ignored, and no information is retained about blocks that are not currently in the cache. This turns out to be surprisingly effective, despite the obvious (self-imposed) poverty of information used by LRU. Clearly, the small amount of information that is recorded turns out to be highly predictive.

Equally important, LRU *ignores* information that usually *doesn't* matter much to its task of prediction. Other, more sophisticated policies have been designed, based on reasonable intuitions, but they have generally failed to outperform LRU, and usually do significantly worse—sometimes disastrously so. LRU turns out to adapt quickly to programs' changing behavior, and is seldom "distracted" or "fooled" by patterns that foil its heuristics. It is dumb, so to speak, but this can be effective because it doesn't "outsmart itself" trying to figure out subtle patterns that may be unreliable predictors for some programs.

It is particularly interesting to study LRU's behavior for looping patterns of memory references, for several reasons:

- Loops are a very common control structure, and often dominate programs' overall pattern of references.
- Loop-like patterns of referencing may arise even when no loop is explicitly programmed, e.g., in coroutine-like uses of bounded buffers, in memory allocators, etc.
- Loops are particularly relevant to LRU's strengths and weaknesses.

The worst case for LRU: A too-large loop. LRU's technique for predicting which pages will be touched soon is quite simple: the time since the last touch to a block is used as a predictor of the time until the next touch.

The bad cases for LRU's predictive strategy are when the time until the next touch to a block is *negatively* correlated with the time since the last touch, such that it systematically evicts blocks that will be referenced soon, in preference to blocks that won't be.

It is easy to construct a worst case for LRU and a memory of size m. Simply loop through $m+1$ blocks of memory, touching each block once before returning to touch any block a second time. (For example, looping through an array of block-sized data objects, reading one value out of each. It doesn't matter whether the items are stored linearly in an array, however.) As long as more blocks than will fit in memory are touched between touches to any given block, LRU will fault on *every* memory reference, and always evict each block just before it is touched again—touching a cache-full of other blocks will always evict every blocks that has not been touched in the meantime.

Notice that for this worst case, it doesn't matter if LRU has m pages at its disposal, or just one, or anything in between. With one page, exactly the same thing will happen, so the other $m-1$ are wasted caching the wrong blocks. It doesn't matter whether blocks are evicted shortly before the next touch, or a long time before it—either way, a miss will occur when it is touched.

This may seem like a fatal flaw in LRU, since simple loops are quite common in real programs, and in fact we do believe this is significant. However, most loops do *not* touch this extremely awkward number of blocks. For any given size of memory, most loops touch either considerably fewer or considerably more blocks than the memory will hold. In the case of "small" loops, where the looped-over data fit in memory, LRU has no problem. All of the blocks will be faulted into memory once at the first iteration, and remain there for any number of iterations of the loop. For loops over very large amounts of data—much more than fast memory will hold, *any* replacement policy will do fairly poorly, even an optimal one.

LRU is competitive with optimal replacement. (Sort of.) The case of "large" loops is clearly bad, but it is significant that for very large loops (much larger than the memory size) *any* replacement policy will do poorly. Consider a memory of size m, and a loop over $2m$ blocks. Even an optimal replacement policy will fault on all of the blocks during the first iteration, and it will fault on half of them at each iteration thereafter. An optimal policy will essentially pick $m-1$ blocks to keep in cache from one iteration to the next, and use one block to cache each of the rest briefly when it is touched. In essence, it will victimize half of the blocks at each iteration, evicting them immediately after use, for the benefit of the other half, which it can then keep in memory indefinitely.

Because of this, LRU is said to be *competitive* with optimal replacement within constant factors of space and time. In general, if we give LRU a constant factor of extra space, we can ensure that its fault rate is also within a constant factor of optimal. For the example above, if we give LRU a factor of 2 in memory, it will not fault at all because the whole loop will fit in memory. More generally, we can always "fix" LRU by giving it somewhat more memory, because the bad cases for LRU in a larger memory are also rather bad cases for optimal replacement in a somewhat smaller memory.

This is reassuring to system designers—LRU may not be perfect, but reasonably good performance (relative to any possible replacement policy) can always be achieved by spending a "small" constant factor more on memory.

On the other hand, this technical notion of (space-time) competitiveness is small consolation to a user of an actual computer with a particular memory size, when a program loops over more data than will fit in memory. The competitive argument requires a constant factor in space to achieve a time bound. For a given memory size, it provides *no useful bound* on how badly LRU may work compared to optimal replacement—LRU may have a 100% miss rate in situations where optimal would simply fault most of the data in once and only fault occasionally thereafter.

(Practically speaking, this may have very serious consequences for the cost of computers. Since LRU's performance may sometimes collapse in the face of a small increase in working set size, it encourages computer owners to simply *buy more RAM* until their worst-behaved program stops "thrashing." A common maxim is that "if it pages, it's broken." This is essentially a cost-maximizing phenomenon—the machine must be able to handle the worst of the programs that are run on it with any expectation of reasonable performance. The machine is loaded with enough RAM for the worst case, because its performance cannot be relied on to degrade gracefully.)

This raises the question of how often programs *do* loop over more data than will fit in memory, and how important those loops are in the overall memory-referencing behavior of a program. Little is known about this.

LRU and multiple looping phases. Most programs' behavior is not dominated by a single loop. Even programs whose behavior consists primarily of loops often do not have a single simple loop that dominates their performance. They may have a variety of phases which are dominated by a simple loop each, but with different-sized loops for each phase. Some of the phases may fit comfortably in memory and cause no problems. Other phases may loop over far more blocks than will fit in memory, and be problematic for any replacement policy, still others are "just right," of an in-between size where another policy might work much better.

As in the case of a mix of whole programs consisting of a single loop each, a single program with a mix of loops of varying sizes may interact well with LRU most of the time, and average performance may be acceptable. Whether this is true often depends on whether *any* loop is larger than main memory, and iterates many times, causing a collapse of virtual memory performance. (The same phenomenon can happen at the level of cache memory for smaller loops, of course, and slow a program down by a factor of several times. If it happens at the size relevant to virtual memory however, the extremely long access times of magnetic disks are likely to cause a much more dramatic performance degradation—perhaps slowing the program by a factor of 100 or 1000, or even more.)

Spatial locality often helps in bad cases. Even within a single program or phase dominated by a too-large loop, several other factors may come into play, making LRU's performance very bad but not nearly the worst case. Spatial locality may help, and nested loops may introduce many memory references that LRU handles nicely.

Spatial locality is often helpful, because a loop over a large number of words may touch a much smaller number of blocks. In the case of an array stored contiguously in virtual memory, accesses to successive elements of an array are usually accesses to the same page. If a page holds 1024 consecutive items of an array that is accessed sequentially, this may reduce the miss rate by three orders of magnitude. In effect, LRU wastes all but one of the block frames, but that one frame (the most recently used one at any given time, not actually a particular frame) does an excellent job of exploiting spatial locality.

Because LRU systematically fails to exploit temporal locality for too-large loops, the miss rate may still be very high—recall that a reasonable miss rate is roughly one in a million—and the program may spend orders of magnitude more time paging than doing "real" computation. In some cases, this may be acceptable; if an infrequently-run program takes hours or days to run instead of seconds or minutes, it may not matter much if the results aren't needed until next week. In other cases, where the program in question must run at a reasonable speed, the only solution is to buy more memory to get acceptable performance.

LRU and nested loops. LRU may do reasonably well for nested loops, where the outer loop iterates slowly and the inner ones iterate more quickly. (Here, it's important that the inner loops iterate over the same data repeatedly—if they iterate over different data, the locality will be much worse.) The blocks touched during the iterations of the inner loops may stay in fast memory between iterations, because they are touched much more often that the blocks touched only by the outer (too-large) loop(s).

In many cases, the inner loops do most of the work and generate far more memory references than the outer loops. In such cases, LRU may work well, but generally performance is still limited by the rate at which the outer loop(s) touch blocks that have been untouched for too long.

Notice that LRU generally does the right thing for short-term patterns in memory references—if pages are touched repeatedly over the short term, they are not evicted from memory. This is its great strength. The mistakes it makes are mostly with longer term patterns, e.g., loops bigger than the memory size.

All other things being equal, this is a good general strategy. If given a choice between systematically making mistakes with respect to long-term patterns or with respect to short-term patterns (and either kind of mistake is equally expensive) then it's generally better to make mistakes on the long-term patterns. **[actually need to discuss skew to make this argument; we happen to *know* that on average, shorter-term patterns are more common.]** Because long-term patterns don't recur as frequently, consistently making mistakes over the short term is more expensive—short-term events simply happen more often. As we will see in discussing frequency-based replacement, neglecting this principle can lead to catastrophic performance collapse.

2.6.3 LRU and other kinds of referencing behavior

[stack-like (mostly LIFO), hot-cold, etc.]

[LRU works well for working sets that fit in cache. Works poorly for working sets that don't quite fit, if referencing patterns are mostly queue-like, but works as well as can be done for LIFO patterns.]

2.6.4 Frequency-based Replacement

An early competitor of LRU was *frequency-based replacement.* The basic idea of frequency-based replacement is to keep track of *how often* different pages are touched, and evict those that are touched least often. The basic idea is that the most important pages are the ones that are touched most often, so they should be kept in the cache in preference to pages that are touched less often.

The simplest frequency-based replacement policy is *Least Frequently Used*, or LFU. An LFU policy keeps a counter for each block in the cache, indicating how many times it has been touched. When a block must be evicted, the on that has been touched the least is chosen.

This seemingly reasonable idea has wide and persistent appeal, but is actually deeply flawed, and frequency-based replacement has performed poorly and erratically in a variety of experiments. Because of this, it is not in general use. We will explore its behavior in some depth, however, for two reasons:

- *It demonstrates important issues.* While LFU works poorly, and this is well known, it is easy to introduce problems like LFU's when designing adaptive algorithms; it clearly demonstrates certain characteristics that replacement policy designers must avoid.
- *Similar issues arise in other guises.* While LFU is not a popular replacement policy, its problems resurface in other contexts, particularly clustering and prefetching policies. We believe that much work on clustering has been misguided, because researchers in clustering have not understood the general kind of problem that LFU demonstrates. Many clustering policies have problems similar to LFU's, which could be avoided. (This will be discussed in a later section.)

Recall that earlier we said that an optimal policy retains the blocks that will be needed soon, and evicts the blocks that won't be. From the replacement policy's point of view, all that matters to any particular replacement decision is how soon each block will be needed—the decision of how *long* to keep it in memory beyond that is not urgent, because it can be made later, independently of whether the block is evicted and faulted in again in the meantime.

The *next* touch to each block is the one that matters, because at that point the replacement policy has no choice but to ensure that it is in memory. If the block will only touched once, it must be in memory, just as surely as if it will be touched a million times.

This is not to say that the pattern of past touches can't be used to predict the pattern of future touches—far from it. At any given time, however, that pattern is only informative if it can be used to predict the *time until the next touch*. In this light, keeping a count of the number of touches to each page seems much less appealing—it only makes sense if the *number* of past touches to a page is very strongly correlated with the *time until* the next single touch—and unfortunately, as experiments have shown, it isn't.

LFU works well for some programs, but often exhibits extremely poor performance, for at least two important reasons:

- It doesn't distinguish between important and unimportant references in a timescale-relative way.
- It doesn't adapt quickly to changing patterns of reference.

LFU and complex (multi-frequency) patterns. Consider a program that touches block A a hundred times as often as block B. If the touches to these pages are distributed evenly, either regularly or randomly, LFU will work quite well. The touch count for block A will quickly rise well above the count for block B, and whenever there's a choice between them, block B will be evicted. On average, that will be the right decision because block A is more likely to be touched next.

On the other hand, it is common that the touches to blocks are *not* distributed evenly. Suppose that block A is touched 1000 times in quick succession every 10,000 time units, and block B is touched once every 1,000 time units. In that case, the bursts of 1000 touches to block A have very different implications than the *spacing* between those bursts. During a burst, the time until the next touch is very short, but between the bursts, it is very long. Ideally, the replacement policy should notice the difference, and never evict block A during a burst of closely-spaced touches. At any other time, though, it should *prefer* to previct block A rather than B, because when block A is idle, it's idle for a very long time.

This is a very severe problem for LFU. Notice that in this case, the count for page A stays very high relative to the count for page B, and LFU will always prefer to evict B, when usually it should evict A. It may therefore make the same mistake many, many times, if touches to other pages force many evictions. It will be battered by extra faults due to this mistake, and never notice that fact and adapt to avoid it. Because LFU is very prone to making serious and *repeated* mistakes, its performance is hard to analyze, but it is generally considerably worse than LRU and often dramatically worse. (It is *not* competitive with optimal.)

LFU and large-scale phase behavior. LFU is also prone to making serious and repeated mistakes due to its inability to adapt quickly to programs' phase behavior.

For example, suppose some set of blocks is touched very often during the intitialization phase of a program, but not at all thereafter. Those blocks' touch counts will go up rapidly for a while, and then stabilize and never come down again. If blocks touched during later phases are never touched as many times, the blocks used during intitialization will be effectively "pinned" in memory, wasting space that could be used to cache the blocks relevant to the later phases. The blocks used later in the program may be repeatedly evicted and faulted on, because the blocks used during an earlier phase were touched so many times that they are difficult to displace.

Attempts to fix LFU. Many attempts have been made to salvage frequency-based replacement. Most of them involve variants of one or both of these two techniques:

- *Filtering out very short-term information.* Here the idea is that high-frequency information is misleading, so that repeated touches over very short intervals should be ignored or given less weight than touches over longer intervals.
- *Decreasing the weight given to older information.* Here the idea is that simple LFU gives too much weight to information about access patterns far in the past. By weighting recent information more heavily, frequency-based replacement can be made to adapt more quickly. Blocks that were touched many times far in the past do not stay "pinned" in memory at the expense of currently-active blocks.

While each of these techniques is a step in the right direction, we believe that frequency-based replacement is fundamentally misguided—these amendments succeed almost precisely to the degree that the resulting policy stops being frequency-based in the classic sense, and starts being guided by a fundamentally different principle.

The deep problem with frequency-based replacement is that its prediction function *predicts the wrong thing*. As the earlier discussion of optimal replacement shows, a prediction function's job is to predict the time until the next touch, *not the number of times that pattern will recur*. Basing time predictions on the number of times something has happened in the past is simply a mistake, except to the degree that it enhances the *reliability* of the *time* prediction.

We believe that a more direct approach is appropriate—rather than simply counting individual events, it is crucial to detect the *relevant patterns* in events. The number of recurrences of the pattern is interesting only in that it bears on the reliability of the prediction.

2.6.5 FIFO

One of the simplest replacement policies is FIFO, or "first-in, first-out" replacement. At a fault, FIFO always evicts the block that has been in the cache for the longest time. This simple policy works rather well, on average—it's miss rate is only about a third higher than LRU's. **[check this number—it might be 20%, or 40%; I don't recall]**

FIFO works reasonably well because most touches are to recently-touched blocks; FIFO ensures that a block that has been faulted into the cache on will stay in cache for a considerable period—until after the eviction of all the blocks that had been there when it was brought in.

Eventually, however, the block will be evicted, even if it has been touched recently—FIFO only notices the *first* touch to a block after it is evicted, and ignores all touches to the block while it is in cache. (A block that is touched once will therefore stay in cache just as long as a block that is touched frequently for the entire time it is cache resident.)

This weakness is not as severe as it may seem—a block remains in cache long enough to avoid *most* misses due to short-term repeated touches. Once it is evicted, it may be faulted on again (once) and then remain in cache for a considerable time. A block that stays active for a very long period will therefore be evicted and faulted on repeatedly, but only occasionally.

FIFO shares most of LRU's strength with respect to simple loops over data that will fit in cache—and its weakness if the data won't quite fit. For a simple loop that fits, FIFO will fault all of the blocks in, and then faulting will stop and the blocks will stay cache-resident and cause no more misses.

For a simple loop that does *not* fit, FIFO will evict the page that has been in memory the longest, which is the one which will be touched again *soonest*, guaranteeing a high miss rate.

FIFO and LRU make the same mistake for simple too-large loops because for such patterns there is little difference between the time of the first touch to a cache-resident block (which FIFO uses) and the time of the most recent touch (which LRU uses)—the block is touched only briefly at each iteration of the loop.

2.6.6 Random

Another simple replacement policy is *random* replacement; when a block must be evicted, it is chosen by some pseudo-random procedure from among all blocks in the cache.

Surprisingly, Random works reasonably well—like FIFO, its miss rate is typically only about a third higher than LRU's. **[get actual number... what is it?]** It is interesting to examine why this is true.

First, consider the fact that most in-cache blocks will either be touched soon, or not for a relatively long time—the distribution of times until next touches is very heavily skewed.

When a random policy evicts a block, it is fairly likely that the block will not be touched soon. [**if we assume some skew in the distribution of touches to blocks, that is... on average, it'll be one of the less-touched blocks or medium-touched blocks, not a hot block.**] In that case, the vacated frame can be used for a considerable time before the block is faulted on again.

Sometimes, however, random evicts a block which *will* be touched soon. In that case, the block is evicted but then quickly faulted back into memory. Once it has been faulted back into memory, it is unlikely to be evicted again soon—the randomness of eviction probabilistically guarantees that many other pages will usually be evicted first.

Random eviction is therefore probabilistically competitive with LRU for working sets that are have very skewed reference patterns. [**Is this what I mean to say? Has this been said somewhere? Is there a proof?**] For working sets that fit in the cache, it will try evicting pages until it evicts one that is not active; on average this takes very few tries. Once the inactive data have been faulted out, the working set is cache resident and faulting stops.

For working sets that do not fit in the cache or do not have a heavily skew, the situation is more complex. If the working set fits in RAM but accesses to those blocks are fairly evenly distributed, Random may evict many frequently-touched blocks, only to fault them in again and try again, repeatedly.

On the other hand, Random does not share LRU's (and FIFO's) severe problem with loops that are somewhat too large to fit in the cache. By evicting pages at random—rather than the blocks that will be touched soonest due to the looping pattern—it ensures that a significant number of looped-over blocks can remain in the cache. Its tendency to evict some recently-touched blocks comes in handy, allowing other blocks to stay in the cache until they are touched again.

2.6.7 Loop-detecting

The very first virtual memory system—for the Manchester University ATLAS, later marketed by Ferranti—used an interesting *loop detecting* replacement policy [Fot]. The idea behind this policy is that loops form an important component of the referencing patterns of many programs, and that loops can be treated properly by keying off of *periodic* touches to pages. Unfortunately, this policy did not work well, and was supplanted by an LRU-like policy. Still, the idea itself is quite interesting and a variant of it may work quite well.

We do not have detailed description of this algorithm, but it appears to have worked roughly as follows: for each block, a record is kept of the time between previous touches, perhaps the last two detectable touches, as well as the time of the last touch.[8]

If accesses are periodic, the time until the *next* touch to a block can be predicted by assuming that the interval between the last touch and the next one will be the same as the time between the last touch and the one before that. The difference between the times of the two previous touches can be added to the time until the next touch to predict the time of the next touch.

It is interesting to consider the difference between this kind of prediction and the predictions made by LRU. Consider the following pattern of touches to a page, where `t` denotes the time at which a replacement decision must be made:

[8] It is unclear to us what mechanism detected this, or whether it could detect all touches to a block. It is also unclear whether it recorded only the last two touches, or some more complex information.

```
       *         *         *
time: -29       -19       -9        t
```

Note that LRU will predict that the time until the next touch is relatively long—the same as the time since the last touch: it's view of the past is simply "reflected" around the present (t) to give an estimate of the future. Since the block was last touched at time $t-9$, LRU predicts it will be touched again at time $t+9$.

The ATLAS policy, on the other hand, will predict that the block will be touched again much sooner—it recognizes the interval between touches (10 units) and sees that another interval is "almost up," so another touch to the page is due soon, at time $t+1$.

This seems like a good principle, all other things being equal, but there are several subtleties.

In a replacement policy, some pages are evicted so that other pages can be kept in the cache; therefore, some pages must be evicted *early*, relative to what LRU would do, so that others can be retained longer because touches to them are expected soon.

The ATLAS policy used its predictions to evict pages early by *noticing when a loop had stopped.* That is, it noticed when its expectation that a page would be touched again was violated—if it expected a touch to a page at (roughly) a particular time, it was assumed that the loop operating on that page had terminated and that the page would *not* be touched again soon.

Unfortunately, memory referencing behavior often consists of more than simple loops, and this heuristic can easily be foiled. Consider a simple case of nested loops:

```
***            ***            ***
 time:                                  t
```

In this case, the inner loop has stopped by time t, but the outer loop (responsible for the repetition of the inner-loop touches) may not have. If the time between the last two touches is used to predict the next two touches, it appears that "the loop has stopped" and the page is no longer active. If the outer loop has not stopped, however, it is very likely that the block will be touched again soon—when the next iteration of the outer loop returns to it.

The problem with the ATLAS replacement policy is that it is not properly *timescale relative.* The relevant periodicity here is the not the periodicity of the inner loop, but the periodicity of the *outer* loop. Very short-duration periodicities do not matter much; the structure of the bursts of touches due to the inner loops is not particularly important. What is crucial is the pattern of touches *at a timescale large enough to be important to replacement decisions.*

We have oversimplified somewhat here, in that we have assumed that memory is large enough that it may be reasonable to hold this block in memory between the iterations of the outer loop. If the iteration interval is *too large*, however, it may be better to evict the page early—the space reclaimed can be used for caching something that will be touched again sooner. In that case, it is better to pay the cost of faulting the block in again later, to avoid more faults in the meantime.

This example illustrates **[...blah blah ...timescale relativity... not too big, not too small, just right...]**

2.6.8 Gap-based replacement

[discuss [Quo94], Phalke, WWOS-IV paper]

2.6.9 OPT or MIN

2.7 Methodological Issues in Replacement Policies

2.8 Toward a Theory of Replacement

2.8.1 Block Histories and Adaptation

2.8.2 Phase Behavior, Aggregate Locality Properties, and Adaptation

3 Prefetching

While many memory hierarchies rely exclusively on demand faulting—waiting until blocks are touched to transfer them to fast memory—some use *prefetching*, initiating the transfer of some blocks ahead of time.

The most common general-purpose prefetching strategy is *sequential* (address-order) prefetching; when a block is faulted on, the following block in address order is also requested. For example, if block number 316 is faulted on, block number 317 is fetched as well. More than one block may be requested, perhaps the next two or three blocks.

More generally, a prefetching policy incorporates a *prefetch prediction* function, keyed to some aspect of program behavior, and issues requests for blocks based on that prediction. Like a replacement policy, the prefetching policy keys off of observable behavior of the program (such as touches to blocks, or faults on non-resident blocks) and uses that behavior to predict future behavior—touches to nonresident blocks that may happen soon.

3.1 Prefetching vs. Large Blocks and Clustering

An alternative to prefetching is to *cluster* (group) blocks or language-level data objects together in slower memory, so that they can all be fetched quickly by a simple "non-prefetching" policy, perhaps using larger blocks. The idea here is to get the *effect* of prefetching using a simple demand fetch policy, by arranging data near each other in memory and fetching larger units. That is, grouping related data together can improve spatial locality, to get much of the benefit of prefetching using a standard demand-fetching policy.

Looked at another way, the normal use of large blocks can be seen as a kind of crude prefetching—after all, "extra" data are fetched when something is faulted on, in the hope that it will also be useful soon.

(Because of this, interpretation of experimental results is not as easy as it might seem, as will be explained later.)

Clustering will be discussed later, in Section 4.

3.2 Programmer-directed vs. Compiler-directed vs. Dynamically-predicted Automatic Prefetching

In this paper, we focus on prefetch schemes which predict future access patterns dynamically based on past access patterns. Other approaches are possible, however, using information about programs from other sources. Programmers may supply directives saying when to prefetch data, based on knowledge of the algorithms used in the program. In some cases, compilers may be able to infer this information, and generate code that issues prefetches, rather than relying on the memory hierarchy itself to make the predictions.

Each of these approaches has merit, but each is seriously limited, as well.

3.2.1 Explicit Directives

In general, programmers are not good at knowing what to prefetch, because few programmers understand issues in locality of reference sufficiently well.

Programmers are likely to make mistakes, and make performance worse, rather than better, if they are not knowledgeable and careful.

[blah blah... need to give programmers a reasonable model, so they can say what they know and let the prefetcher evaluate it and decide what to do, rather than having them simply say what to fetch: similar to overlays-vs-VM; the runtime system has information that the programmer doesn't, e.g. memory size, so the programmer shouldn't overcommit at programming time.]

3.2.2 Compiler-directed Prefetching

[compiler-directed prefetching works best either for very short-term access patterns—up to about a hundred instruction cycles ahead—or for extremely regular patterns, like blocked arrays. Doesn't work well for anything that can't be determined statically by relatively local analyses of the source program... compilers just aren't good at the "big picture," but runtime systems can be... mix of both is likely to be best... fwd. ref striping and blocking in clustering section.]

3.2.3 Dynamic Prediction

[why dynamic prediction is necessary... regularities in data that aren't known at compile time, nonlocal properties that are difficult for compilers to infer, etc.]

3.3 Block size and Fetch Policy

When considering the use of prefetching, it is important to keep in mind that normal demand fetching using moderate block sizes already shares important characteristics with prefetching—some data are fetched "speculatively," because they are near the demanded data. A prefetching policy therefore can have three different kinds of effects:

- *increasing the fetch size*, by fetching more blocks at a time,
- *increasing flexibility in fetch size*, by allowing more blocks to be fetched at some times, but not at others,
- *increasing flexibility of* what *is fetched*, and
- *increasing flexibility of eviction*, by allowing prefetched blocks to be evicted independently of faulted-on blocks.

3.3.1 Effects of Increasing the Fetch Size

When comparing prefetching and demand fetching policies, it is therefore important to separate out these effects. If a demand-fetching policy and a prefetching policy are compared *using the same block size*, the results may misleadingly favor one or the other simply because a given program may work best with a particular *fetch size*.

Consider a program with fairly good spatial locality, such that for a particular memory size of interest, say 4 MB, the optimal fetch size is 8KB. If we just compare a simple one-block-lookahead policy and demand paging, using 4 KB virtual memory pages in both cases, the prefetching policy will naturally look better: it exploits spatial locality and can fetch the same amount of data with half as many seeks.

What this seemingly head-to-head comparison does *not* tell us is whether the benefit is due to the differences between demand fetching and prefetching, or just due to a larger fetch size. It is not unlikely that nearly all of the benefit from prefetching would also result from using demand fetching with pages that are a larger size, say 8 KB.

To really be able to compare these two policies, we must at least compare prefetching using 4 KB pages to demand fetching using both 4 KB and 8 KB pages. A one-block-lookahead policy may sometimes fetch one page, and sometimes two, so we should bracket its average fetch size in comparing it to a demand fetch policy. If prefetching does not work better in both cases, the apparent advantage of prefetching in one case may not be robust. It may simply be due to the fact that the tested program has higher spatial locality than the block size can fully exploit, and can benefit from an increase in fetch size in either way.

In trying to evaluate prefetching policies in a general way—as opposed to making assumptions about the block size—it is therefore necessary to find a fair baseline. One good baseline is the performance of demand fetching with the*demand-fetch-optimal* block size.[9] For each test program and each particular memory size used, the page size should be adjusted to min-

[9] We invented this term for this paper. **[pointers to prior statements of this concept would be appreciated]**

imize total (latency + transfer) costs for the demand fetching scheme.

Then the prefetching policy should be used with two or more block sizes that bracket this size; if it works better than demand fetching with the same block size, that means that the prefetch policy does not unduly increase transfer costs when it increases the fetch size.

If prefetching works better in both cases, then we can conclude that prefetching has definite advantages over demand fetching, above and beyond simply increasing the fetch size.

Even if a particular prefetching policy does not robustly improve performance over a range of block sizes, caution should be exercised in interpreting this "negative" result. There are many possible variations in prefetching policies, and a simple negative result does not necessarily mean that "prefetching doesn't work." It means that further study is required, to determine why the tested policy doesn't work, and whether the problem can is fundamental, or can easily be fixed.

3.3.2 Effects of Flexibility in Fetch Size

One advantage of prefetching is that *in principle* it should be possible to adjust the fetch size dynamically, by fetching more or fewer pages at a time, to adapt to particular workloads' spatial locality characteristics. Prefetching may therefore be able to make a memory hierarchy's performance more robust, by adjusting the fetch size to a particular workload—rather than just picking a fetch size expected to work well "on average." (This adaptation might be dynamic, during a program run, or it might be based on entire executions of a program, or it might be based on the overall job mix a computer faces.)

This is a very complex and poorly understood topic, however, and most prefetching policies don't seem to make any (intentional) attempt to do this.

To properly evaluate this effect, it is necessary to find a "representative" set of programs—which is a very difficult subject—and find a set of "reasonable" block sizes, such as the demand-fetch optimal block size for the entire test suite. If, on average, prefetching works better than demand fetching for fixed block sizes and a range of programs, it demonstrates the benefits of prefetching per se.

3.3.3 Increasing Flexibility of What is Fetched.

One advantage of some prefetching schemes is that they may fetch different data than would be fetched simply by having a large block size.

[blah blah... often harder than it looks... sequential prefetching is usually cheaper than random access prefetching—especially for disks, but also for silicon memories to a lesser degree because of the way memory units work—fetching blocks in same row of the 2D memory array is faster than fetching blocks in a different row, because an enire row is usually latched and can be read from without reading the row from the main array again.]

3.3.4 Increasing Flexibility of Eviction

A related advantage of prefetching may be to allow prefetched data to be evicted independently of data that are actually touched. For example, consider a demand-fetching memory that fetches 8KB blocks, and a sequential prefetching memory that fetches 4KB blocks using one-block lookahead. These policies are comparable in terms of what they fetch, but may treat data very differently after they are fetched. Suppose, for example, that only one half of each 8KB page is actually touched. In that case, the demand-fetching memory must cache 8KB blocks, even if half of the block is never touched.

In contrast, the prefetching memory can evict the 4KB prefetched block that goes untouched, while keeping the 4KB block that was actually touched in memory.

Cutting losses due to bad prefetches. A simple example is when the demand-fetched block is touched repeatedly over a long period of time, and the prefetched block is never touched at all. Eventually, the prefetched block will be evicted, and the demand-fetched block can stay in the cache indefinitely, as long as it keeps being touched often enough. Thus a prefetching memory has the ability to "cut its losses" in the face of mistakes by its prefetch predictor.

This raises an interesting and poorly understood question of how long prefetched blocks should be kept in memory in hopes that they will be touched soon. Perhaps prefetched blocks that aren't touched *very* soon after they are fetched should be evicted, to avoid polluting the cache. But then, perhaps not—it may be that a prefetch which doesn't pay off immediately is still likely to be good, because it will be touched *soon enough*. Perhaps unsurprisingly, the question of whether a block will be touched "soon enough" is timescale-relative. If the cache is very small, a prefetch may have to pay off very soon to pay off at all, because the memory occupied by the prefetched block could be put to better use. If the cache is large, it is not too expensive to keep the block around for a longer while.

The prefetch time-to-payoff distribution is relevant to this question. If it is very heavily skewed toward short payoff intervals, then it is worthwhile to evict prefetched blocks fairly quickly if they aren't touched. For example, it may be that prefetches usually pay off very soon or not at all. In that case, evicting prefetched blocks if they go untouched for a short period will cut losses without reducing prefetch performance by much. The successful prefetches will still be successful, and the unsuccessful ones won't waste much cache space.

If the time-to-payoff distribution is less heavily skewed, it may be worthwhile to keep blocks in memory for a longer time if the memory is large, but a shorter time if the memory is small.

(If the distribution is not significantly skewed, prefetching may simply not be worthwhile; prefetched blocks are likely to either be evicted before actually being touched, in which case the prefetch did no good and some harm, or the prefetched blocks may be touched while in memory, but only after they have indirectly caused more misses by occupying blocks that could have been put to better use in the meantime.)

Unfortunately, little is known about skew in the time-to-payoff distribution; this issue has not generally been studied directly. Most studies simply present bottom-line performance results for entire policies, without separating out the reasons for their success or failure.

Independent eviction. Prefetching may be advantageous relative to fetching larger blocks even when *both* blocks are touched, so that the prefetch is successful, but one block remains active longer than another. If we simply used demand fetching with blocks twice as large, the resulting large block would be a large unit that must be either kept resident or evicted. With prefetching, it is possible to evict either block independently of the other, if it does not keep getting touched over as long a period.

3.3.5 Fetch Size vs. Overall Memory Size, and Timescale Relativity

[This section is now somewhat redundant... should it be moved up, or should part of the stuff in the previous section be moved after this?]

Another consideration in choosing a fetch size is the total size of the memory. For small memories, small blocks may be especially desirable, because *a small memory size effectively reduces spatial locality.*

Recall that spatial locality is really a spatio-temporal phenomenon—fetching a larger amount of data at a miss is good if the "extra" data will be touched "soon." But how soon is soon? That is, when does it pay off to fetch extra data, and when is it better not to, and fetch the data (in smaller blocks) as needed?

The answer to this depends on three things:

- how soon the extra data will be touched (if at all), and
- whether the memory it will occupy could be put to better use in the meantime,
- whether sufficient bandwidth is available to satisfy the prefetch requests without saturating the communications channel (bus or disk adapter).

In the case of a small cache, space is typically quite precious—only a small amount of very-recently-accessed data can be held in the cache, and items typically don't stay in the cache very long. Bringing anything into the cache requires evicting something else that is very likely to be accessed fairly soon. In general, bringing the extra data into the cache will pay off only if that extra data is touched *sooner than whatever data it replaces in the cache.*

For a large cache, this problem is less severe. An item that is evicted usually hasn't been touched for a long time, and is unlikely to be touched again soon; evicting it to make room for extra data brought in by a fetch is rather less dangerous. (If the extra data still go untouched while they are in the cache, however, they will of course tend to pollute the cache and increase the miss rate.)

Notice that whether the data go untouched is timescale-relative. For a large cache that holds blocks for a very long time, the extra data may go untouched for a considerable period, but then be touched while in the cache, saving a miss. For a smaller cache, the same data may already be evicted by the time they are touched, causing misses—after uselessly wasting space in the cache, and likely causing misses on other items that could have been successfully cached.

Luckily, such considerations are usually only critical for very small memories, which have relatively few block frames. Most modern memory hierarchies have several hundred or even several thousand block frames at each level, so the choice of block size is more dependent on transfer and latency times and the cost of maintaining address translation mappings. (This may not be true for small first-level caches, or for very small caches used in the implementation of CPU's, such as translation lookaside buffers, etc.)

3.4 Overlap and Bandwidth Limitations

Two important factors can decrease the effectiveness of prefetching. One is a lack of *overlap* between fetching data and normal program execution—if a block is not requested far enough ahead of time, the program may demand the block before it is available, and have to wait for it despite the fact that it was prefetched. Another is a lack of *bandwidth*; if fetches (including prefetches) occur too closely together, there may not be enough communication bandwidth between fast and slow memory, and the limiting factor of performance will be the rate at which the data can be transferred.

3.4.1 Overlap

For a prefetch to be effective, the prefetch must occur far enough in advance of the program's actual access to the requested block. If a prefetch is initiated, but the program immediately attempts to access the data, the program must wait for the block almost as long as if it had simply faulted on the block.

The *overlap ratio* is the fraction of the fetch time that is successfully overlapped with other computation—the program's execution of instructions that operate on registers or data in the cache, before actually attempting to access the prefetched data.

The overlap ratio depends on features of the actual memory hierarchy in question, particularly the latency of misses. It is therefore a measure of the potential effectiveness of prefetching for a memory system with particular parameters. If the latencies are very high, the overlap ratio is likely to be low—a smaller fraction of the latency can be masked by prefetching ahead of time. (This will be discussed in detail below.)

A more general measure of the performance of a prefetch policy is the *prefetch time-to-payoff distribution.*[10] This is the distribution of times between the initiation of prefetches and actual touches to prefetched blocks; it just says how many prefetch predictions occur how far in advance of actual touches to the predicted blocks. This distribution can be interpreted relative to systems with different miss latencies.

For example, if a prefetch prediction occurs 20 instruction cycles in advance, and we assume that a prefetch is immediately initiated, then it may be an entirely successful prefetch if the miss latency is less than 20 instruction cycles—or a partly effective prediction if the prefetch is 40 instruction cycles, in which case we may be able to reduce the cost of a miss to 20 instruction cycles by prefetching 20 cycles in advance.

Notice, however, that the prefetch time-to-payoff distribution is *not* generally independent of the memory size or block size, because whether a prefetch is issued at all generally depends on whether a block is memory resident. (For a small memory, it may not be resident, and a prefetch may be issued; for a large memory, it may be resident and no prefetch is necessary.)

The prefetch time-to-payoff distribution is enlightening in two ways. The first is obviously that it tells, for any given miss latency, how much latency cost may be masked by prefetching. The second is that it tells how long memory may be occupied by prefetched pages waiting to be touched. If prefetched pages are touched before they are evicted, they may be "successful," but if they go untouched for a long time, they may also incur a cost by tying up memory, in effect polluting the cache.

All other things being equal, the best prefetch predictor would prefetch just far enough in advance to mask the maximum amount of latency cost, but not so far in advance that prefetched pages unnecessarily tie up cache space that could be put to better use. The ideal prefetch time-to-payoff distribution is modal, with a mode at a point just beyond the actual miss latency. Blocks will be fetched "just in time" to get the maximum benefit in avoiding stalls by overlapping fetching with computation—but no so far ahead

[10] We made up this term; it seems to us an obvious idea, and may be used elsewhere. Pointers to prior use of this concept would be welcome.

as to squander fast memory.

Even this is something of an oversimplification, however. It assumes that if a prefetch is initiated far enough in advance to mask the latency, it will actually do so. This may not be correct, however, if prefetches are clustered too closely together in time.

3.4.2 Bandwidth Limitations

Even if prefetches are issued "far enough" in advance to mask latency, there may be problems when too many prefetches are issued in quick succession. Rather than latency being the bottleneck *bandwidth* may become the bottleneck.

Suppose, for example, that the prefetch predictor makes two predictions at successive instruction cycles, each correctly predicting a touch to a different block, 20 cycles in advance. If the miss latency cost is 20 cycles, including 10 cycles for latency and 10 cycles for transfer, then only the first prefetch will be entirely successful—the second prefetch will have to wait 10 cycles for the first one to finish, before the bus becomes available again to transfer the block.

If more than two prefetches occur in quick succession, this problem may be exacerbated. At some point, the bus bandwidth becomes saturated, and prefetches can only be satisfied as fast as blocks can be transferred. At this point, the problem is really that there simply isn't enough bandwidth to satisfy the programs' needs for data. The only way to solve this problem without increasing the available bandwidth is to space the prefetch predictions out over more time. If a program touches a lot of memory over a short period of time, prefetches may have to occur very far in advance to avoid bandwidth limitations. In general, this is quite difficult.

This issue has not been studied in any depth. In some cases, detailed simulation studies have shown prefetching policies to be ineffective, and the conclusion has been drawn that there was not enough overlap. We believe that in at least some of these cases, overlap may not have been the problem—unrecognized bandwidth limitations may have been the bottleneck. For a given memory system, it doesn't matter much which is occuring, because either way the system runs slowly. In terms of memory hierarchy design, however, the difference is crucial—rather than suggesting that prefetching doesn't work, it may suggest that bandwidth limitations are more important than previously realized.

3.5 Prefetch-always vs. Demand Prefetching (Prefetch-on-miss)

Prefetching policies are often categorized by whether prefetches only occur at actual misses, or may occur at any time, whether a miss occurs or not. In high-speed silicon cache memories, each has its advantages. In virtual memory systems, on the other hand, usually prefetch-on-miss makes more sense.

The reason for this is that latencies in virtual memory systems are huge, and the overlap ratios are correspondingly tiny, but another trick can be used to effectively increase overlap for some prefetch policies.

Rather than overlapping normal program execution with prefetching, it is possible to overlap *two fetches*, because of the peculiar characteristics of disk access times.

At this point, it is important to note that disks are not really "random access" devices, in the sense that main and cache memories are. The cost of accessing a block is very strongly dependent on which blocks have just been accessed. In particular, accessing blocks that are widely separated on the disk generally incurs a seek—the read head must be repositioned—and often a significant cost in rotational latency. In contrast, reads of sequential disk blocks are often vastly cheaper—once a block has been read, reading the next block is much, much less expensive. Reading a random block costs a seek and rotational latency, but reading the following block simply requires waiting for it to pass under the read head.

Because disk block access times are far from uniform, some prefetching policies are far, far cheaper than others. If the block we want to prefetch happens to be the next block on the disk, the read head is already in the right position, and all we have to do is continue reading. Transferring an extra block may only take about a millisecond, while the block passes under the read head, rather than taking several milliseconds to seek to another track and wait for the desired block to come under the read head.

The most common kind of prefetching policy for disks is therefore *sequential* prefetching of disk blocks. When a disk read occurs, the prefetch policy simply requests the next block on the disk, too.

Rather than overlapping computation with I/O, this effectively overlaps the *latencies* of multiple I/O operations: in effect, we combine the seeks for multiple blocks into one, and the additional "seeks" are free. For the extra blocks, only the transfer cost remains.

[**this happens in silicon memories as well,**

but to a less dramatic degree, because of row latching in 2D memory units...]

3.6 Replacement Policy for Prefetched Blocks

[how long to keep prefetched blocks? Prefetch time-to-payoff relative to replacement interval]

[added stuff to earlier section about this... elaborate here, or bag it?]

4 Clustering to Improve Spatial Locality

An alternative to conventional prefetching is *clustering*, which is the grouping of related data objects or blocks within larger blocks to improve spatial locality. The best-known kind of clustering is grouping of data objects such as records within virtual memory pages or disk blocks. It is also possible to group smaller blocks within larger units relevant to slower memory, such as grouping several virtual memory pages within larger units of disk transfer to improve paging performance.

By grouping related data, spatial locality can be improved so that normal demand fetching or sequential prefetching works better. This is especially appealing in reducing disk seeks, since disk latencies are so high that overlaps are usually low, and nonsequential prefetching is unlikely to yield significant benefits.

4.1 The Ubiquity of Clustering

While clustering may seem unusual, it actually is not. It arises in obvious guises in systems such as object-oriented databases, but in less obvious guises in virtual memory systems, file systems, memory allocators and garbage collectors, and compilers and linkers. In general, any mechanism that allocates storage (or arranges how data are stored) performs clustering, intentionally or not, because it decides which items go where. Many such mechanisms are intentionally designed to cluster things according to some principle or other, but some are not—the fact that they perform clustering goes unrecognized.

Many virtual memory systems group pages together dynamically during paging, writing out groups of pages together. This is usually not recognized as clustering in the traditional sense, but this mechanism could be exploited to improve spatial locality if done well.

Most file systems perform some form of intentional clustering, if only to keep files stored mostly sequentially to optimize sequential reads and/or writes. Many also cluster file metadata (such as directory information) so that it can be accessed without fetching the contents of files. Log-structured file systems [RO91] provide tremendous flexibility for clustering, although their potential has hardly been explored. (Work to date has concentrated primarily on decreasing the cost of writes, or of avoiding fragmentation of the disk, but an LFS's ability to store any data anywhere on the disk opens up many possibilities for improving spatial locality, reducing read costs as well [SKW92].)

Conventional memory allocators (like C's `malloc()` and `free()`) also implicitly perform an important kind of clustering, simply by choosing where in memory to put objects when they are initially allocated. A good understanding of the principles of clustering could lead to the design of better allocation algorithms with improved spatial locality, or to better choices among the many available allocator algorithms [WJNB95].

Copying garbage collectors [Wil] perform clustering at each garbage collection, grouping objects according to the reachability traversal by which the collector identifies the live objects.

Compilers and linkers perform clustering of program code and data. A compiler groups procedures together in some order, often the order that they are declared in a source file. A linker groups compiled modules together into executable files, arranging larger units into an order that will be reflected in memory when the program is loaded.

4.2 A Unified View

Clustering is poorly understood, perhaps even more poorly than prefetching. We believe that the issues in clustering are very similar to the issues in prefetching, and that this point has generally been overlooked. Issues of timescale relativity have not generally been addressed. A wide variety of clustering techniques has been used in various systems and in various simulation studies.

The literature is incoherent; some strategies have been tried in some contexts (such as copying garbage collectors), and others in other contexts (such as object-oriented databases), but few techniques have been broadly applied. Some techniques have only been experimentally evaluated using synthetic data, which (we will argue) is unsound.

We believe that research in this "area" has been hampered by the fact that workers in clustering are spread through several different technical communities (operating systems, programming languages, data bases, etc.) and seldom read or criticize work in related areas. A general model of clustering has not been adopted or validated, and relevant work in related areas is frequently overlooked.

In this section, we introduce a new model of the clustering problem, focusing on issues of timescale relativity and the satisfaction of multiple clustering goals for varying access patterns. Some of these ideas are implicit in some prior work, but have not been fully developed and are not widespread. There has been no unified presentation that clearly defines the clustering problem, and important issues have usually been overlooked in most of the mainstream clustering literature.

After presenting our intuitive model of the clustering problem, we then survey various basic kinds of clustering techniques, and applications of those techniques to particular kinds of systems.

4.3 Goals of Clustering

One goal of clustering is clearly to group related data together, so that fetching large blocks is effective; a fault on an item within a large block should be a good predictor that other items (data objects such as records, or small blocks) in the same block will be touched "soon." This corresponds to prefetching in that good clustering, like prefetching, can reduce miss costs directly.

On closer examination, however, it is clear that other issues come up in clustering, as they do in prefetching, but in subtly different ways.

In some kinds of clustering, like the grouping of small data objects within virtual memory pages in a conventional memory hierarchy, individual objects *cannot be evicted independently*, as prefetched blocks can. We will call this a *simple clustering* scheme. A simple clustering scheme cannot "cut its losses" the way a prefetching scheme can, by evicting prefetched items early if they are not actually touched, or if they remain active for different amounts of time after being loaded into fast memory.

A good prediction function may therefore be more critical for simple clustering than for traditional prefetching—ideally, the clustering policy should never have to cut its losses, meaning that the items in a block should *always* be accessed together, not just fetched together and soon touched. If it ever happens that some items in a block are touched at very different times than other items, that implies that some items will have to be kept in memory unnecessarily, and waste space.

The real goals of clustering are therefore:

- *To together group items that are accessed together,* to improve the effectiveness of simple fetch policies with a large fetch size.
- *To* separate *items that are accessed differently,* so that accesses to some objects do not force others into memory, or force them to stay in memory, polluting the cache.

These two goals *can* be in conflict. If items are sometimes accessed together, but sometimes not, should they be clustered together in a block, or not? In general, this seems like a hard problem, and one that depends on timescale relativity in subtle ways. Luckily, timescale relativity also comes to the rescue, and shows that it is not usually as hard as it might seem at first.

To achieve good prefetching, we need two things:

- *A good prediction function*, that predicts which objects will be accessed together with a fair degree of reliability, and
- *A good strategy for clustering*, based on that prediction function, which will minimize wasted I/O and cache space usage.

We will address the second issue first. For the moment, assume that we have a magical prediction function that reliably predicts actual access patterns, and we are trying to come up with a good clustering.

This is actually a fairly good approximation of some clustering problems, such as the offline grouping of program code and data based on profile information derived from traces of actual program runs. (In such a situation, the prediction function just predicts that future access patterns will strongly resemble those of the training runs; this prediction may not be entirely valid, but it is often the best available information.)

4.3.1 Timescale Relativity in Clustering

To begin with, we must have a timescale-relative notion of what it means for items to be accessed "together." For any given memory size and miss cost, "together" means within a period of time that is comparable to the timescale of cache replacement.

For example, for a small high-speed cache memory, blocks may be considered to be accessed "together" if touches to them occur within a few thousand instruction cycles of each other. If they are touched a million instruction cycles apart, they probably shouldn't be considered to be accessed together, and clustered together, because that would amount to issuing a prefetch prediction far in advance of the actual touch: bringing the predicted item into fast memory at that point is likely to waste space, and may do no good at all, because the block may be evicted before the predicted item is touched.

On the other hand, for the purposes of clustering virtual memory pages, the very same access pattern may be interpreted as meaning that the items *are* accessed "together." A difference of a million instructions is small at the timescale at which virtual memories operate—fetching things a millisecond sooner or later makes essentially no difference to the effectiveness of prefetching. What matters is whether the data will be accessed "soon" on a timescale that is usually seconds or minutes, and may be longer. (Keep in mind that a normal virtual memory has several thousand pages, and can only fetch or evict a hundred or so per second, even when it is completely I/O-bound.[11])

While the issue of timescale relativity complicates our analysis slightly, it actually makes the real problem easier—it means that for any given memory configuration, we can ignore most of the information about the access patterns. For a small memory, we can ignore large-scale patterns, and just focus on averaged short-term statistics. For a large memory, we can ignore short-term patterns.

For the time being, assume that we are talking about clustering objects into pages for a simple demand-paged virtual memory system. In this case, we need to group together objects that are typically accessed within (roughly) a few seconds of each other.

For any reasonable size of memory, an important point to notice is that *it is not important to group particular pairs of related objects into the same page.* For example, suppose we have several objects, A, B, C, and D, which are always accessed together and in that order when any of them are accessed. Suppose that we can fit two of them into a block. We might do the obvious thing, and group A and B together in one block, and C and D together in another. This would clearly be a good clustering.

On the other hand, we could also put A and D together in one block and B and C together in another. Either way, when we access the items in order, we will incur two faults, and we will bring them all into memory very quickly. It doesn't really matter much how they are grouped into pages, as long as the *set* of items that are accessed together is grouped into a *set* of blocks.

Unless the set of pages is very large, we can assign the items to blocks *in any way we like* with very little difference in performance.

How large is a large set? That is, how much freedom do we really have? Consider the fact that most levels of a memory hierarchy usually have several hundred, or more likely several *thousand* blocks. Splitting a group of closely related objects across several pages may cost essentially nothing—a few pages of data may be fetched slightly earlier than necessary, but if those objects are also touched reasonably soon, it doesn't matter.

Consider a more realistic example, where we have a few thousand items, known typically to be accessed together, which we must cluster into a hundred blocks. In the worst case—if we assign the set of items into the set of blocks in the worst possible way—we will waste less than a hundred blocks by "prefetching" some items earlier than necessary. For a cache with thousands of block frames, this is unlikely to make a signficant difference in cache performance; we'll waste at most a few percent of our cache space. Equally important, if the items are known to be accessed at roughly the same time, we'll waste that space very briefly—just until we get around to touching the "prefetched" items.

As long as we know that the set of items will be accessed at *roughly* the same time, and the set is small relative to the size of memory, how we group particular objects into particular pages is almost irrelevant. The "prefetching" due to clustering will be successful, because the "extra" objects in faulted on pages will be touched fairly soon.

[11] We have oversimplified slightly here, in a way that will be rectified later. The relevant measure of time is not wall-clock time or CPU time; it is relative to the number of blocks touched. Thus two items are touched "together" if the number of other blocks touched in between is small relative to the memory size.

This means that we have considerable leeway in grouping of objects that are accessed together—we need not focus on grouping closely related *pairs* of objects into single pages, or even closely-related small groups of objects onto a few pages. That's a good thing, because we must deal with items that are not *always* accessed together, and try *not* to group them into the same blocks.

4.3.2 Keeping Semi-together Items Semi-together

So far, we have assumed that we are clustering together items that are known to always be accessed together, if at all. This is only half the problem. The other problem is keeping items that are sometimes accessed *differently* apart. We should avoid grouping items in the same page if some of them may be accessed without the others being accessed at roughly the same time. (Again, the "same time" is relative to the size of memory, and the information in needn't be precise in general. For virtual memories, it can be very coarse.)

For example, suppose we refine our example of a few thousand data items being clustered into a hundred blocks. Let's say that 20 blocks' worth of those items are always accessed if any of them are, but the other 60 blocks' worth are sometimes *not* accessed at those times. So, for example, some program phases may access the entire set of items, but others only access the 20 pages worth of items that are always of interest.

This is representative of an apparently common situation in a variety of programs. For example, in a database-like system, one program might iterate over a set of commonly-used records, while another might iterate over that same set *plus* a set of related records that give more information about those items.[12]

As another example, consider file systems that separate directory information from the contents of normal files. Some programs may simply traverse directories, while others mostly read file contents, and others do a combination of these things.

We believe that this is common in many (if not most) nontrivial, data-intensive systems—that is, most programs for which locality is important. Most programs perform a variety of different operations on overlapping sets of data, using indexing data structures to keep the different characteristic access patterns efficient.

In this example, we want to do two things:

- Keep the entire set of 100 blocks' worth of items "together," in the sense that they are grouped into a set of 100 blocks, and
- Avoid intermingling the 20 blocks of items that are accessed by both access patterns with the 80 blocks of not-always-accessed items. That is, the 20 blocks of often-used items should be kept especially together, in a small subset of the blocks that keep the overall group together.

If we don't satisfy the second goal, we're likely to have to fault in all or nearly all of the 100 pages whenever we access the 20 pages worth of frequently-used items. If we do satisfy it, we can still satisfy the first goal. The goals are not actually in conflict, even if it may seem that way at first.

4.3.3 An Example Clustering Problem.

Consider Figure 5. In this picture, an index data structure (whose structure is not shown) holds pointers to a series of n data objects a, each of which has pointers to several auxiliary objects, b, c, d, and e. For simplicity, assume that the clustering algorithm deals well with indexing structures and does not intermingle the indexed items with the internal structure of the index itself. (This will be elaborated later.) Just consider the two-dimensional set of objects ($a \ldots e$ by $1 \ldots n$).

Many clustering algorithms will notice that each object a is directly connected to the corresponding objects b, c, d, and e, and cluster these directly-reachable

[12] This principle is well-known in physical design of relational database systems. A logical relation is often split into multiple "physical" tables (data structures such as sequential files or B+ trees), keeping only those attributes together that are always or almost always accessed together by all common queries.

Similar issues arise in the implementation of interactive programming systems, where the different kinds of information about varibles are often intentionally separated into separate tables; a compiler may examine variable name strings, as well as value fields and other information, but a running program may only access the value fields. For example, in Common Lisp implementations, it is common to separate the parts of a "symbol" table into a set parallel vectors (1-D arrays), rather than using a single vector of records. Among other things, this separates the fields that are likely to be accessed only by the compiler (such as variables' name strings) from those that are likely to be accessed by running applications (such as variable "binding cells" that hold variable values).

More generally, parallel arrays are often used in array-intensive programs, rather than arrays of records, to separate out fields that are accessed by different phases of a program.

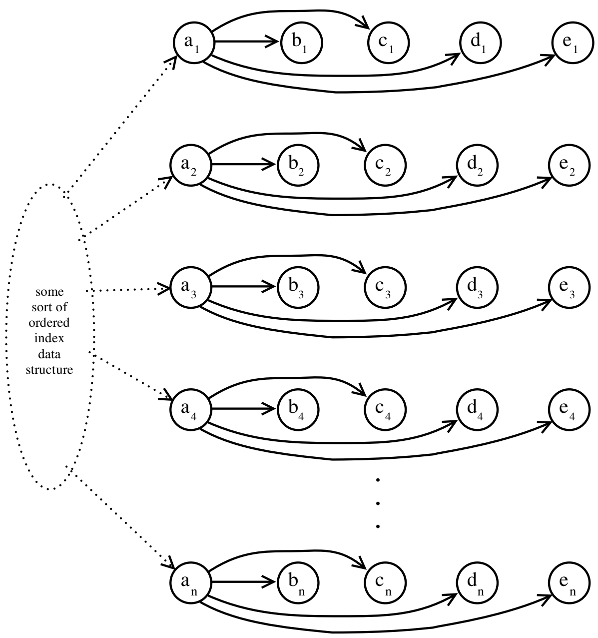

Figure 5: An Example Data Structure to be Clustered.

objects with it. This seems reasonable, and may in fact be the right thing to do, depending on access patterns in the program that operates on the data—but often it is the *wrong* thing to do.

Suppose that we have the following frequent access patterns:

- The index is traversed sequentially, and each object a is touched, but none of the objects it references is touched.
- The index is traversed sequentially, and each object a is touched, and its pointer fields are traversed to touch its b, c, d, and e.

If these are the only important access patterns, the objects a should be grouped sequentially, in index order (more or less), and the other objects should be grouped *separately* from the a's, but in a similar order.

Notice that even if the accesses to the objects are not strictly sequential, in key order, this may still be a very good clustering. If the index searches are nonrandom, and similar keys are often used—to look up objects that are "near" each other in the key ordering, useful spatial locality will result. Bringing one object a into the cache will also retrieve a set of other objects a with similar keys. Bringing one object of the other kinds will also bring in objects of those types *whose a's have similar keys*, and are therefore likely to be accessed soon.

When is this likely to be a *bad* clustering? The clearest case is when accesses to a's exhibit no locality in terms of the index keys (e.g., truly random accesses to a's), and each access to an a is followed by an access to its b, c, d and e. In that case, each fault on an a may be followed by a fault on another page or more to retrieve the associated objects. On the other hand, if the random accesses to a's do not usually involve accesses to the related objects, keeping a's separate may actually be beneficial.[13]

For one likely access pattern, separating the a's from the b's is likely to be very beneficial, and for another, it is harmless. For another, however, it does harm.

This example illustrates three important points:

- Clustering algorithms should not be too local and greedy. Grouping objects by local connectivity may group together objects that should be *separated*, because they are not always accessed at about the same time.
- Multiple access patterns may not cause goal conflicts in terms of clustering, as when grouping a's separately from their related objects for the first two types of access patterns.
- Sometimes goal conflicts *do* arise, as in the case where index probes may be random *and* random accesses to a's usually include pointer traversals that touch the related objects.

4.4 Offline vs. Online Clustering

A clustering algorithm may be used online, to recluster data dynamically according to actual access patterns. For example, a virtual memory system might record the order in which pages are actually touched, and incrementally regroup objects during normal operation to dynamically tune the system to its workload.

Many clustering systems are *offline*, however. Rather than performing clustering while programs run, some information is used to group items together *before* programs run, and during execution the clustering cannot be changed. This is common in normal

[13] For example, if the a's could all fit in memory, but the entire data structure could not, then this would allow caching the entire randomly-accessed data structure.

memory hierarchies, which provide little support for fine-grained regrouping of objects within virtual memory pages.

Offline clustering clearly has less ability to adapt to dynamic patterns of accesses than online clustering. On the other hand, the potential of offline clustering has hardly been understood, much less explored thoroughly, and it may well be that cheap offline techniques can provide much better locality than systems that don't pay attention to clustering issues.

A particular weakness of offline algorithms is that they generally cannot cluster items based on past patterns of access to those items if the items only exist at runtime. They only have data about references to objects during past program runs, not during a future run, and can only reorganize "persistent" data such as file data, saved system heap images, and code and data in executable files.

On the other hand, this weakness may not be as crippling as it appears; it may be possible for an offline system to abstract away from individual objects and learn what works for clustering *kinds* of objects. Lessons learned from prior experience can then be exploited in a (non-adaptive) online algorithm.

Simple examples of this include experimenting with different memory allocation algorithms linked into a program, so that a good one can be chosen for actual use, or trying several module orderings when linking a program, to find one that exhibits better behavior during testing runs. More advanced approaches require the use of more advanced instrumentation and analysis to observe the behavior of the program, and guide the search for a likely clustering technique.

4.5 Sources of Information to Guide Clustering

Whether online or offline, real clustering systems usually rely on partial information to guide clustering, using heuristics in the prediction function. Obviously, any real and general clustering system is unlikely to have exact and detailed information about *future* access patterns, and must rely on past behavior or the implementor's intuitions.

If detailed information about access patterns is not available, the clustering algorithm clearly must settle for weaker knowledge, and rely more on heuristics to provide a "prediction function" that can be used for clustering.

Among offline clustering algorithms, some require some information about the actual pattern of dynamic accesses during "profiling" or "training" runs. Others rely on simple heuristics, such as grouping objects based on their pointer connections and/or their types. Others rely on declarations provided by programmers.

Even among online clustering algorithms, however, the amount of information available for clustering may be severely limited. For example, in most systems, there is no easy or affordable way to record information about the order of accesses to individual small objects. An online clustering system may therefore have to rely on heuristics to estimate the prediction function.

In some cases, an offline clustering algorithm may have richer information than an online one. For example, it may be possible to instrument a program in fairly expensive ways to gather information during training runs, but such expensive instrumentation would be prohibitive for runtime use in normal operation. (For example, a program might be run under a machine code interpreter to gather very detailed information, but it would run very slowly; normal users would not generally pay this cost in actual use.)

4.5.1 Access Sequences

The best kind of information for guiding clustering is actual access sequences, such as reference traces, which show in detail the ordering of touches to individual items to be clustered. Such data can be analyzed in a timescale-relative way to determine a good clustering for the intended memory configurations (e.g., large memories or small).

In an offline algorithm, records may be kept of recent access patterns, and this may be use to dynamically reorganize data. This information might be in the form of reference traces gathered during training runs, or even during normal operation. The storage and I/O cost of such information can be very large, however.

For an online algorithm, efficiency is expecially important, so it is especially important to keep the overhead low, and record only the most predictive information. Some systems reorganize data periodically, according to the order of first accesses to an object within a period; all other information about access order is lost.

4.5.2 Profiles

Some clustering systems are based on simple *profile* data, which records the frequencies of certain operations—for example, how often a particular pointer field of a particular kind of object is traversed. This can be used to inform the clustering algorithm of the importance of particular kinds of events during execution.

Profile data can be misleading, however, because it is usually not timescale-relative. Most profile-driven clustering schemes are based on simple heat, either object heat (which objects are touched the most times) or link heat (which pointer links are traversed the most). Typically, link heat is used to cluster objects greedily, preferentially grouping objects linked by hot links. Less greedy algorithms may be used, using heuristics to group objects so that the total heat of links across block boundaries is minimized [?].

Object or link heat may not be a good metric of the importance of clustering objects together, however, for two reasons:

- *It is not timescale relative.* Heat is not timescale relative (Sections 2.5.1, ??) and may be misleading. The importance of a reference to an object (or a link traversal to an object) depends strongly on the temporal pattern of references. Many references to an object over a short period of time may have little importance for clustering, while a few references over a longer period of time may matter much more.

- *It does not distinguish between overlapping working sets.* Objects may have equal heat, but if they are accessed at very different times, they should generally not be clustered together. Similarly, if two objects are reached from the same object *via comparably hot links*, but the links are traversed at very different times, the objects should not be grouped together. (Section 4.3.3.

For example, if we note that we traverse pointers from a's to b's 10 times as often as pointers from a's to c's, does that suggest that b's should be grouped with a's, in preference to grouping c's with a's. Maybe so, but maybe not.

Recall that the problem with LFU replacement was that the count of individual touches to a page might have little to do with its actual importance for caching. Likewise, the frequency of a particular kind of pointer traversal may have nearly nothing to do with the importance of that kind of pointer link for clustering.

Suppose, for example, that links from a's to b's are usually traversed during phases where little data is actually touched, and locality is not particularly important because everything fits in the cache. (For example, the same pointers may be traversed repeatedly over a short period of time, during CPU-intensive iteration over a fairly small set of objects.) Suppose that in contrast, pointers from a's to c's are traversed during very data-intensive phases that iterate over very large volumes of data.

It may well be that the frequency of link traversals is positively correlated with their importance for clustering, but this has never been demonstrated, and there is every reason to question this assumption. Many programs consist of different kinds of phases, including very regular operations over large amounts of data and much less regular, CPU-intensive operations over much smaller amounts of data. This suggests that the correlation between link traversal frequency and importance for clustering could be *negative* in some of the most crucial cases. Simple clustering by links (without traversal frequency weights) may work as well or better.

4.5.3 Reachability via Pointer Links

Many systems cluster data by reorganizing objects according to the pointer links between them. For garbage collected language implementations, this is often done during copying garbage collection. For object-oriented databases, it may be done during occasional reorganizations of persistent data.

A reachability-based reorganization starts from some set of "root" pointers, from which all objects are directly or indirectly reachable. (In an object database, this might be a top-level directory object, which holds pointers to major indexing data structures. In a garbage-collected language implementation, this might be the set of pointers in local and global variable bindings and registers.

Typically, reorganization happens during a traversal of the data structures reachable from the roots. Some exhaustive graph traversal algorithm is used, such as breadth-first or depth-first, and objects are moved to new storage as they are reached by the traversal. (When an object that has already been reached and moved is reached again, the traversal is short-circuited at that object, and the pointer is simply updated to point to its new location.) Since objects are copied to new storage as they are reached by the algorithm, the traversal order determines the clustering.

Simple graph traversals. Two common traversal orderings are depth-first and breadth-first, but these both have potential weaknesses. A depth-first algorithm may tend to plunge far into the reachability graph, traversing many pointers, grouping objects that are only distantly related by pointer links. A breadth-first traversal is more "even handed," but tends to decompose the reachability graph into layers. At the first layer (near the roots), it tends to group siblings together, but as successive generations of descendents are reached, it may group more and more distantly related cousins together.

An alternative is to use a *hierarchical decomposition* [WLM91] of the reachability graph, which tends to pack pages with the nearest descendents of an object, e.g., the first few levels of a tree, and then recursively do the same for the descendents of the objects on that page, packing each subtree into its own page if possible.

For simple trees, hierarchical decomposition ensures that the path from a root to a leaf is as short as possible, in terms of the number of blocks visited. This closely resembles a B-tree, in that the upper levels of the tree, when viewed as a block, effectively act as a single larger node in a multiway search tree; this node just happens to be internally indexed as a lower-arity tree. Likewise, subtrees below this "node" (block) are similarly packed, putting as many levels as possible in one block.

Since linear lists are degenerate trees, a hierarchical decomposition will pack successive elements of a list into a block. (This also happens with depth-first and breadth-first traversals.)

Simple graph traversals tend to cluster related objects together, in that objects with pointer links between them are typically more likely to be touched together than objects that are not linked—typically, an object is reached by following pointer links from other objects. While this is much better than a random organization [Bla83], it still may not be particularly good, because some pointer links may be far more important than others in terms of locality.

A simple graph traversal clusters objects via pointers that *may* be traversed, but those pointers may *not* be traversed, or may be traversed in some phases and not others. This has four important weaknesses:

- Some objects may be reachable via multiple paths, and a typical (greedy) clustering will group objects together according to the first link encountered by the traversal, which may not be the most important one.
- Some links may not be traversed much, and grouping according to those links may give "unimportant" objects equal weight, at the expense of grouping more important objects together.
- Some links may be traversed during some phases, and others during different phases. Grouping objects according to the structure of the reachability graph may intermingle working sets that are separate at run time, wasting cache space when only a subset of the objects are actually needed in cache.
- Some links may be from indexes with extremely poor locality, such as hash tables, and grouping according to those links may make locality much *worse* by grouping together objects which are accessed in randomized ways [WLM91].

Type-sensitive traversals. To avoid some of the problems with blind graph traversals, a reachability-based algorithm may be enhanced by making it sensitive to the *types* of objects that are encountered during a traversal. Objects of different types reachable from the same object may be clustered *apart*, in the expectation that different kinds of objects may be accessed differently [LWM92]. Some types may be treated specially, such as hash tables, so that objects are preferentially clustered according to links that are likely to yield better locality [WLM91].

Less greedy traversals. Most reachability-based clustering schemes cluster the reachability graph as though it were a tree—the first pointer to each object is used to determine the object's placement, and subsequent pointers to the same object have no effect on placement.

It would also be possible to take into account the sharing of subgraphs of the graph; objects reachable via multiple paths might be clustered apart from those reachable via only a single path. This might tend to separate objects used during multiple kinds of phases from those used during a single kind of phase, especially if the different paths are from different large indexing data structures.

4.5.4 System- and Application-specific Declarations

While the ideal clustering system would be fully automatic, it is also possible to exploit programmers' knowledge of application behavior to find a good clustering. Programmers often know that certain sets of objects are likely to be used together, while others are only used during distinct phases.

Link weight declarations. Some systems provide a mechanism for the declaration of the importance of links. For example, when defining a class or record type, the programmer may specify that a certain pointer field should have strong weight in determining a clustering, and others should not.

Explicit clustering directives. In other systems, objects may be assigned to clustering groups when they are allocated, or the programmer may give hints that certain objects should be clustered together.

A common way of doing this is for the underlying storage mechanisms to maintain separate storage pools, which may just be sets of pages. (These are variously known as "areas," "segments," "arenas," "files,"or "heaps.")

In several systems, a programmer can explicitly specify which storage pool to put an object in when the object is allocated. Alternatively, the programmer may give a hint that an object should be clustered near some other object, and it is the allocator's job to attempt to find a satisfactory clustering based on the hints.

These explicit clustering directives may be augmented with other clustering techniques. For example, the programmer may specify which objects belong in which storage pools, but the system may augment that coarse grouping by clustering objects within a pool according to a reachability-based scheme.

4.5.5 Discussion

4.6 Some Clustering Schemes

4.6.1 On-the-fly Reorganization of Virtual Memory Pages on Disk.

In its normal operation, a normal virtual memory system detects the order of accesses to pages in a timescale relative way—at a page fault, it is known that a page is being touched for the first time since it was evicted. A virtual memory system typically maintains an approximation of an LRU recency queue as well, and therefore can detect the order of *last* accesses to pages over the timescale relative to the cache.

It is natural to attempt to exploit this information, by grouping virtual memory pages on disk in an order that will improve locality for future accesses. By ordering pages on disk in a way that reflects dynamic access patterns, and using sequential prefetching, it may be possible to fetch more useful data per seek.

In [BS76], Baer and Sager attempted to apply this principle in simulations of a virtual memory system. Their simulated virtual memory system regrouped data at eviction time, grouping the four least-recently-used pages together in a block.

Unfortunately, the results were disappointing. However, on examination of their experimental design, it appears that they may have chosen unrealistically small memories, and inappropriately large block sizes. (This was apparently due to the fact that their test programs simply didn't use much memory.) Their memory sizes were generally less than 100 pages, and sometimes much less; in terms of current systems, this means that the page size was very large with respect to the memory size, and the system was already fetching too much data at each seek. (Recall that modern memories typically have several hundred or even several thousand blocks at each level of the memory hierarchy.) Naturally, any extra prefetching is likely to exacerbate this problem, so a negative result for prefetching may be pessimistic.

For programs with larger data sets, and a memory with several thousand blocks, such a dynamic reorganization may work much better. Examining Baer and Sager's data, there appears to be a trend of improving performance (relative to a non-prefetching policy) as memory size increases. Extrapolating to a more realistic ratio of memory size to block size, it appears that Baer and Sager's technique may in fact be worthwhile.

4.6.2 On-the-fly Reorganization of Objects in an Object-oriented Memory Hierarchy.

More recently, the MUSHROOM group at the university of Manchester have simulated the effects of clustering individual objects in an "object oriented" memory hierarchy. This system uses the same basic principle as Baer and Sager's, but relies on the use of a high-speed cache architecture that caches arbitrary-sized objects. This allows the reorganization of individual objects within pages, rather than just reorganizing pages within larger units of disk transfer.

The MUSHROOM simulations produced encouraging results, although the tested workload consisted primarily of small Smaltalk programs. Further experimentation is needed to assess the effects of this promising technique for other (and larger) workloads.

The main drawback to the MUSHROOM system is that it relies on an unusual hardware architecture, which supports the relocation of individual objects. In general, this kind of architecture is more expensive than traditional memory hierarchy designs. The improvements due to clustering must be weighed against the cost of a novel hardware design.

4.6.3 On-the-fly Reorganization of Objects during Incremental Copying Garbage Collection.

Incremental copying garbage collectors based on Baker's algorithm naturally reorganize data in memory, in the order that they are reached either by the garbage collector or the running application program.

An incremental collector traverses reachable data in small units of traversal work, interleaved with small units of application execution. A copying collector relocates objects as they are reached by the collector, compacting them into a contiguous (or mostly contiguous) region of memory. In Baker's scheme, pointers touched by the running program are also incorporated into the collector's traversal, and immediately relocated. This requires a "read barrier," which is a special sequence of instructions executed at each pointer operation, to detect whether a new object has been encountered. (Lisp machines had special hardware support for the read barrier, to avoid the extra instructions at a cost in hardware and/or microcode.)

In [Whi80], White pointed out that the read barrier could be exploited to reorganize data according to a program's actual access patterns. The garbage collector's normal traversal of data structures could be suppressed during most of a garbage collection cycle, so that the only relocation was due to the program's accessing pointers and the read barrier's copying them to the new heap region. (This can be augmented by having the normal background traversal copy data into a different region of memory.) Objects reached first by the running program will thus be relocated and clustered in the order in which they're actually touched.

For Baker's original algorithm, locality is still quite poor, because a simple garbage collector tends to touch a large amount of memory before it is compacted—compaction is "too little, too late" to avoid the major problem of failing to reuse memory promptly [Wil]. Generational garbage collectors can greatly reduce this problem [?], however, and the dynamic reorganization principle is applicable to incremental generational copying collectors.

Courts [Cou88] applied this principle to the generational collector of the Texas Instruments Explorer (a Lisp Machine), with good results.

The difficulty with this incremental copying scheme is that the cost of the read barrier may be prohibitive on standard hardware, slowing program execution by several instructions at each pointer operation. Other incremental collection techniques may perform better in terms of raw performance [WJ93, Wil], but do not provide a "hook" for reorganizing data according to access patterns.

4.6.4 Profile-driven Reorganization of Disk Cylinders

Heat-based reorganization has been used in disk storage, to perform a simple kind of very coarse-grained clustering and reduce seek distances. Entire disk cylinders are exchanged, putting the hot cylinders near the center of the disk head's throw, so that most seeks will be to locations near the center. (This is not the center of the disk, but the center of *radius* of the disk, between the spindle and the edge of the platter, over which the disk head seeks.)

The best-studied technique for disk cylinder reorganization simply puts the hottest cylinders near the center of the head's throw, so that seeking from one hot cylinder to another is cheap, and seeking to cooler cylinders is more expensive [VC90]. This is called an "organ pipe" arrangement, because a histogram of disk seeks looks like the pipes of a church organ—tall columns in the middle, with decreasing column heights toward the sides.

Timescale relativity and "heat" in disk reorganization. The effectiveness of the organ-pipe arrangement has widely been interpreted as evidence that heat-based clustering is likely to be effective in other contexts (e.g., in [?]). However, it is important to realize that the "heat" used in disk reorganization is generally *not* simple heat in the sense that we have used it so far.

The statistics that are used for disk reorganization are usually based on *disk seeks*, not simple block accesses. The profiles used to guide reorganization are not profiles of touches to blocks by application programs, but of touches that *miss the main-memory cache*. The use of miss profiles rather than touch profiles improves timescale relativity. Frequent touches to cached blocks—which are not generally important in terms of locality—do not get recorded.

Replicating hot blocks.

4.6.5 Reachability-based Clustering in Copying Garbage Collectors

4.6.6 Reachability- and Type-based Clustering in Copying Garbage Collectors.

4.6.7 Allocation-order and Size-based Clustering in Conventional Memory Allocators.

4.6.8 Profile-driven Organization of Code and Statically Allocated Data at Link Time.

4.7 Discussion

5 Architectural Considerations

5.1 Basic Memory Hierarchy Organization

5.1.1 Interactions Between Levels

5.1.2 Basic Structure of High-Speed Caches

5.1.3 Basic Structure of Virtual Memories

5.1.4 Disk I/O

5.2 High-speed Cache Memories

5.2.1 Associativity

Fully associative caches.

Set-associative caches.

Direct-mapped caches.

Associativity and Speed.

Associativity and Locality.

Victim Caches.

5.2.2 Write Policy and Write Buffering

Write-back.

Write-through.

Write-around.

5.2.3 Split Instruction and Data Caches

5.2.4 Subblock (Sector) Caches

Write-validate.

5.2.5 Virtual vs. Physical Indexing and Tagging

5.2.6 Prefetching

Simple prefetching.

History-based prefetching.

5.3 Virtual Memory

5.3.1 Translation Lookaside Buffers (PTE Caches) and Traps

5.3.2 Implementing Replacement Policies

FIFO.

Protection Bits and Segmented Queue Approximations of LRU.

Reference Bits and Clock Algorithms.

Dirty Bits and Write Policy.

5.3.3 Variable-Space Policies and Process Scheduling

Working sets and allocation of memory to processes.

Thrashing.

Job scheduling to avoid thrashing.

5.3.4 Page Tables

Multilevel Page Tables

Inverted Page Tables

5.3.5 Memory-mapped files

5.3.6 Shared memory and mapping

5.3.7 Sharing and Protection Issues

5.4 Disk Storage Management

5.5 Some Novel Memory Systems

5.5.1 Flash RAM

5.5.2 Distributed Caching and Distributed Virtual Memory

5.5.3 Compressed Caching

6 Toward a Deeper Understanding of Reference Locality

[this is partly redundant now... this section will elaborate earlier ideas based on intervening sections on architecture, etc., pulling things together better]

[As noted earlier,] designers of memory hierarchies generally assume some degree of temporal and spatial locality, but the *causes* of locality are seldom examined in detail. Locality of reference is a complex function of regularities in data, data structures and algorithms used to manipulate the data, and compiler, linker, and allocator decisions that determine where those data are in memory at run time.

6.1 Where Does Locality Come From? (revisited)

6.1.1 Clustering

6.1.2 Checkpointing

6.1.3 Allocation and (Re-)Initialization

Initialization Misses. **[So far, we have usually assumed that all misses cost the same amount.]**

This is not always true, however, as we noted in **[the architecture section]**. A particularly important case where it may not be true is initial misses, the first time a block is touched. At first glance, it may seem that initial misses must always cost what any normal miss costs—after all, a block is generally not brought into the cache until the first time it is touched. This is not necessarily true, however, depending on why the miss actually occurs.

If the miss occurs because data are actually being faulted in from slower storage, then generally the miss costs the normal amount. On the other hand, if a block is being used for the first time, and initialized with *new* data, it may not cost anything at all—rather than faulting the blocks old contents into the cache from slower storage, it may be possible to "create" an empty block "out of thin air," in the cache. This requires that the cache distinguish between a touch to a pre-existing block and a creation of a "new" block, and treat each appropriately. In either case, however, a block frame is required to store the new block, and this usually requires the eviction of an older block cached there; if that block is dirty, its contents must be written back to slower storage.

This kind of optimization is performed by most virtual memory systems. When a program requests more virtual memory from the operating system, the operating system knows that this is virtual memory that has never been touched before, and does not have any "old" contents. The virtual memory system may therefore reserve a block frame in the cache, and simply initialize it with zeroes, or leave its old contents in place on the assumption that the program will overwrite them as new data objects are created in the page.[14]

(Typically, new memory is not requested from the operating system directly by an application program. The application program usually uses a library (e.g., containing the C library functions `malloc()` and `free()`, which manage free storage). When the allocation routine cannot satisfy a request for block, it request more virtual memory from the operating system, typically in units of one or more pages, and subdivides that memory to accomodate the application's requests.)

Some high-speed cache memories include special features that accomplish much the same thing, and in fact optimize for a more general case. These *write-validate* caches (which we will discuss in more detail later) notice when any miss is due to a write, rather than a read. In that case, they create the block "out of thin air" in the cache, but keep track of which words (or some other sub-unit) of the block are written to

[14] For security reasons, many operating systems zero-fill pages, to keep old contents (which may belong to other processes) from being visible. As an optimization, the virtual memory system may keep track of whether the old contents belong to the same process, and only zero-fill pages if they do not.

before they are read. If a word is written to, it is marked "valid", and subsequent reads will read the new value. If a word is read before being written to, its old contents are faulted into the cache from slower storage. (Generally, all of the "missing" words of the block are faulted in at this time, to avoid faulting on individual words.)

Interestingly, the write-validate optimization works in cases where the application does not request "new" memory, but simply overwrites blocks of data. Consider a program that uses a large, statically allocated array, repeatedly filling the array with values and then performing some computation over those values. Each time it goes through the array, filling it with new values, the write-validate optimization comes into play. The old values are not loaded into the cache at the first write to each block; the blocks are simply created in cache and "initialized" with their new values without bothering with the old values.

At a certain level of abstraction, it may seem that the write-validate optimization is doing something fundamentally different (as well as more general) than what a virtual memory system does in creating a new page *ex nihilo*. In the case of the virtual memory optimization, a new virtual page is being created, and presumably initialized with new language-level data objects (e.g., heap-allocated objects created by calls to `malloc()`). In the case of write-validate caches and statically-allocated language-level objects, the optimization is detecting the assignment of new *values* into *existing* objects.

If we examine these cases at a *higher* level of abstraction, however, the cases may seem more similar. When a statically-allocated object is reinitialized, this often corresponds to the creation of a *new* data set, implementing a *new* conceptual, application-level object. The fact that a language-level object is reused is secondary in such cases; interestingly, the write-validate optimization often collapses the language-level differences—allocating a "new" object vs. reusing an old one for new data—and treats them the same way because they are fundamentally the same thing.

6.1.4 Indexing

6.2 Varieties of Locality

6.2.1 Hot/Cold Locality

6.2.2 Mostly-LIFO locality

6.2.3 Mostly-FIFO locality

6.2.4 Address-sequential locality

6.2.5 Repeated-sequence locality

[Timescale relativity: e.g., sequence of misses rather than sequence of touches.]

6.2.6 Similar-sequence Locality

[literal vs. comparable repetitions—moderate differences in order make no difference at most scales.]

[higher-order patterns, e.g., strides, LRU-LRU transform]

6.3 Locality at Different Levels of Abstraction.

6.3.1 Locality in Data.

6.3.2 Program-level Locality of Reference.

[incl. allocation vs. re-referencing]

6.3.3 Memory-level Locality of Reference

6.4 Effects of Allocator choice

6.4.1 Memory Reuse

6.4.2 Effects on Clustering

6.4.3 Nonmoving Allocation

6.4.4 Garbage Collection

6.4.5 Nonmoving Allocation

6.4.6 Compaction and Regrouping

6.5 Data Structure and Algorithm Choice

6.5.1 Arrays

mapping by indexing—regularities in what's indexed map to regularities in what's accessed.

row-major vs. column-major

blocking

6.5.2 Lists

Hot-cold temporal locality may work if only front of list is often accessed. But timescale relativity matters a lot: if any elements further down the list are touched very often at all, all intervening elements will be touched.

Spatial locality is also crucial: if the list is not grouped, locality will be bad. If list is grouped but in wrong order, locality will also be bad—you'll fault in most of the list even if you only touch relatively few elements.

6.5.3 Trees

[At the logical level (accesses to program objects linked into a data structure) trees may or may not have excellent locality, depending on the key sequence used to probe the tree. If the key sequence is sequential, or there are strong key-space localities, locality is likely to be excellent. If the keys are effectively random, locality will naturally be poor.]

[Language-level locality in trees vs. address locality: if tree is not grouped in good order in memory, bad news. Grouping the tree like a b-tree, with the upper levels together on a page, and then the upper levels of subtrees on separate pages, is probably best [WLM91]. Conventional binary tree rebalancing may systematically destroy locality if no reclustering is done.]

6.5.4 Hash Tables

[Hash tables generally have terrible locality at their own scale. Randomization systematically causes bad locality—that's a hash table's job—but hash tables can have big secondary effects, either by affecting clustering in disastrous ways, or by affecting algorithms that iterate over tables and touch the indexed data structures in randomized order. This can usually be avoided by keeping the table entries in key order or insertion order, and using a hashed index into that ordered collections [WLM91].]

[At a fairly fine granularity, spatial locality may be strongly affected by the strategy for collision resolution. Rehashing is often thought to be good, because it disperses colliding keys, but this naturally causes accesses to distant parts of the table. Simple linear search for an empty slot may have much better spatial localit, tending to keep a single search within a single block. [?]]

6.6 Checkpointing

6.6.1 The Importance of Checkpointing

6.6.2 Checkpointing at the Data Structure Level

6.6.3 Checkpointing at the Virtual Memory Level

6.6.4 Interactions of Memory Allocation and Checkpointing

6.7 Effects of Programming Model

6.7.1 Sharing vs. Copying

File I/O vs. Persistence

Explicit Freeing vs. Garbage Collection

6.7.2 Process Models and Interprocess Communication

6.7.3 Persistence

7 Algorithmic Analyses for Data-Dependent Algorithms

7.1 Tree Algorithms

[Blah blah...]

Perhaps an example is in order here. One well-known recent invention is the Sleator-Tarjan splay tree, a kind of self-adjusting binary trees presented in [ST85]. Using an amortized analysis, Sleator and Tarjan have shown that splay trees' worst-case time performance is at least competitive (within a constant factor) with that of any conventional binary tree, for a sufficiently long sequence of inputs. This result is interesting in itself, and has received considerable attention. On the other hand, the motivation for splay trees is not primarily to provide logarithmic bounds on average access times, which is easy using a variety of

tree structures—it is to provide a highly *adaptive* data structure which can "beat the odds" on average, by exploiting certain common regularities in many kinds of real input data. Splay trees exploit a *recency* property known to occur in several kinds of data sets—if a key occurs at a particular point in a sequence, it is more likely to be seen again soon than other keys.

Whether or not [the dynamic optimality conjecture or whatever] can be proven, it is interesting to ask several questions:

- **How common is this regularity in real data sets? When can it be expected to occur, and when not? For what kinds of applications does this make splay trees attractive?**
- **What other regularities do splay trees exploit?**
- **What other regularities are common and easily exploitable, perhaps by a variant of splay trees? What features can be combined so that an algorithm can exploit any or all of several common regularities, for robustly good performance? Which of these features is combinable with good worst-case performance (as in the case of splay trees) and which entail taking a risk of very poor performance to exploit expected regularities?**

[Splay trees exploit a kind of spatial locality, in the space of the key values used to probe the table. On average, they move nodes along a search path halfway toward the root, making future accesses to things with near-by keys faster...

This may be optimal within constant factors for any kind of tree. How is this possible, when optimal online replacement—a seemingly similar problem—is not possible? Because for tree lookups, the cost of a adapting to an access when it happens is comparable to the cost of adapting to it before it happens—figuring things out ahead of time isn't as critical, because the cost of reorganizing the tree is similar either way.]

7.2 Graph Algorithms

7.3 Compression Models

8 Data Regularities, Algorithmic Regularities, and Locality of Reference

8.1 Multiple Levels of Organization

9 Analytic Models

Locality and caching studies usually involve detailed simulation experiments, to show how well a proposed memory hierarchy design would work for some real applications, with the expectation that this will resemble its performance for other applications. This is the soundest kind of memory hierarchy study, because it can take into account many possible interactions between actual program behavior and the behavior of caching policies.

Sometimes, "analytic" (mathematical) models of cache performance are developed, concisely summarizing some of the tradeoffs in cache design for some small set of design parameters—most often, the cache size, associativity, and block size. Often, these mathematical models are parameterized in terms of "program behavior," and the intent is to show how different kinds of workloads will interact with different cache configurations. Certain relevant characteristics of real workloads are measured, and it is shown that a mathematical model based on these measures can predict cache performance reasonably well—for a *small* set of design variations.

When choosing between analytic models and experimental evaluation, the first thing to realize is that *the analytical models are extremely weak*. They are generally based on a fairly superficial analysis of program behavior and its implications, and gloss over many interesting and important issues. Many are based on low-order Markov models (described later), which capture only short-term sequences of events and are systematically biased toward certain unrealistic kinds of longer-term behavior that is directly relevant to memory hierarchies.

Because the analytic models are so limited, extensive simulation is often used to evaluate important designs, and in any *exploratory* research involving novel ideas about locality.

[need more empirical exploration of principles, not just bottom-line comparisons of specific designs.]

9.1 "Analytic" Models?

So-called "analytic" models of program behavior are not always truly analytic. They are often "analytic" in the weak sense that certain consequences follow naturally and inevitably from certain axioms (and possibly input data). Unfortunately, the axioms are sometimes poorly-motivated, and the relevant inputs poorly understood. Analytic models may be quite sound mathematically, but as *scientific models*, they are usually quite simplistic, and far from analytic. Program behavior usually usually can't beunderstood "from first principles" in many interesting senses relevant to memory hierarchies.

This is not to say that some analytic models aren't useful for some purposes. If a designer wants a rough estimate of the effects of adjusting some fairly well-understood tradeoff—e.g., doubling the cache associativity vs. doubling the number of sets—an analytic model may be helpful. (On the other hand, simple interpolation in tables of known cache behavior may do about as well.) Often, the mathematical formulae of analytic models are best viewed as a concise representation of empirical data—a handy way of doing curve fitting. Such a model may also be very useful for *theory testing*; whether the model *fails* to predict important phenomena correctly may guide the search for explanations.

Sometimes models embody principles that can be used to extrapolate reasonably beyond the known data, but often they do not, and in any case such extrapolation is intrinsically risky—it is guesswork, and should be recognized as such and done very carefully. Often a highly mathematical presentation of a model conceals the fact that the model is simply not realistic.

For important decisions, and *especially when the design is novel*, there is no substitute for detailed simulation experiments using real traces program behavior. The analytic models embody so many assumptions that they are invalid outside a fairly narrow, well-understood region of the design space. For example, most models do not address locality and timing issues relevant to prefetching *at all*, or only address those issues relevant to the very simplest kinds of sequential prefetching. If one of these models is used to predict the performance of a sophisticated prefetching policy, *the answers are probably going to be wrong,* and if they're not, it may be due to blind luck.

9.2 The Roles of Models

The development of "analytic" models of program behavior has two intertwined themes:

- An attempt to summarize empirical results succinctly, without necessarily understanding their underlying causes; and
- An attempt to show that known regularities have inevitable effects on memory hierarchies, and thus to "reduce" observed regularities to the workings of underlying mechanisms.

In examining models, it is important to keep these very distinct ideas in mind. Much modeling work mixes the two, and often rightly so. Certain aspects of program behavior are understood well enough to allow phenomena to be explained clearly and analytically. Other aspects are understood approximately, and models may include "fudge factors" based on observed regularities that are not clearly understood. This is common in scientific modeling, and is not necessarily a flaw.

Sometimes, modelers attempt to do without fudge factors, and derive powerful models from very simple ideas. Often these attempts are abject failures, whether or not that failure is recognized. Sometimes the attempts are reasonably successful, within certain very narrow limits of applicability, but grandiose overgeneralizations are made about program behavior. Frequently, a highly mathematical presentation conceals a serious lack of understanding of the underlying phenomena.

In reading the following, the main fact to keep in mind is that program behavior is generally not well-understood, so any attempt at a grand unified model is likely to be premature.

9.3 Common Weaknesses

Most current models of program behavior suffer from one or more of the following limitations:

- *Narrow scope.* The most accurate models tend to only reliably model program behaviors relevant to certain well-understood cache designs, such as simple fully-associative LRU caching. As more features are added to the cache (set associativity, prefetching, etc.), the models' weaknesses become increasingly severe. In some cases, empirical results can provide reasonably effective fudge factors over a slightly broader range of well-known designs, but any really novel design is likely to depend on realities that the model simply does not capture.

Little attempt has been made to model the effects of higher-level aspects of program behavior and language implementations. The models do not provide any way of predicting performance for programs written in different styles or using different fundamental algorithms, or for different language implementation strategies (e.g., compilers and allocators). Such variations are sometimes implicitly encoded by parameters that account for grossly different observable behaviors, but the relationships between these parameters higher-level causes are not identified. The models encode the low-level "what" of memory-referencing behavior, but not the higher-level "why" of where that behavior comes from.

- *Failures of timescale relativity.* Many models attempt to derive large-scale behavior of programs from known facts about small-scale behavior. It is assumed that if the small-scale behavior is modeled correctly, and the correct larger scale behaviors will follow naturally. In general, neither assumption is true. (Unless, of course, a model is so detailed as to amount to running an actual program in simulation, in which case it stops being an interesting "model.")

  The better models *start with* program behavior at the scales relevant to the phenomena being modeled, generally without explanation of how that relates to smaller-scale behavior. This is often a good strategy, because the origins of large-scale behavior are often unknown. A shallow empirically-based model is likely to be much more accurate than a model derived from "first principles" that are simply wrong.

- *Dangerous independence assumptions.* Many models assume that unknown behaviors are simply random, or effectively random for the purposes of the model. Sometimes, this assumption is safe within the model's scope—given its intended applications, patterns in the unknown variables may be known not to matter.

  Often, however, the assumption that events are effectively random—and hence independent of each other—is important, and may be systematically false. In reality, may matter whether certain kinds of events happen in bursty or repetitive patterns, or whether they are correlated with other kinds of events.

- *Lacking or incorrect validatation.* Some models are quite well validated, and known to be accurate for a variety of workloads over some intended scope. Other models have never been validated against real workloads at all, or only against a tiny sample of workloads (e.g., one or three programs).

  Sometimes validation has been attempted using the same assumptions made by the model. Rather than using real workloads such as reference traces, statistics from real workloads are used to validate *some* of the assumptions of the model, and the other assumptions are assumed to be reasonable and correct. (A variant of this problem is the use of synthetic programs, which will be discussed **[later]**.)

9.4 Simple Markov Models

Many models of program behavior are *Markov* models of some one or another, because Markov models are simple and often mathematically tractable. Unfortunately, Markov models are often systematically bad at modeling the aspects of program behavior that are relevant to memory hierachies.

A Markov model is based on a kind of randomized process that exhibits behaviors according to certain statistics. Known regularities are characterized statistically, and a random process is used, with a weighting function, to generate those behaviors in the appropriate proportions.

Markov models are not "just random," of course—if they were, they wouldn't be useful for modeling much of anything at all. The weightings of events according to statistics ensures that certain regularities will be exhibited over any long enough sequence of events.

Markov models characterize regularities using a *finite state machine*, i.e., a fixed graph of nodes representing "states," connected by directed edges representing state transitions. The system is assumed to be in one of a fixed (though possibly very large) number of states at any given time, and the transition from

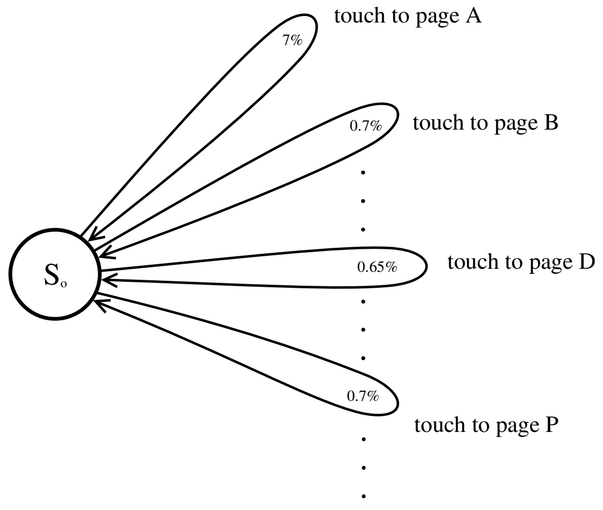

Figure 6: An IRM (zeroth-order Markov) model.

each state to the next is random, but weighted by the statistics.

9.4.1 The Independent Reference Model—a trivial (zeroth-order) Markov model of memory referencing

The simplest kind of Markov model is a zeroth-order Markov model, where there is only one state, but there may be any number of state transitions representing events, all of which return to that state.

For example, suppose we want to model the frequencies with which different pages are touched, but nothing else (such as patterns in the sequence of touches. We can have a simple start state, and arcs that represent touches to different pages. We weight the arcs with the probabilities of touches to each page.

Figure 6 shows such a zeroth-order Markov model.

We can generate a statistically accurate trace from this model by simply looping to pick among the arcs in a random way, but weighted by their probabilities. For the chosen arc, we emit a reference to the corresponding page. Then we loop to do it again, for as long as desired. The resulting trace will (probabilistically) exhibit a realistic proportion of touches to each page, but will probably not resemble the real trace in any other interesting way.

This model of memory referencing is called the *Independent Reference Model* (IRM) [?], because touches to pages are independent of each other—they can occur in any order, with a fixed probability of touching any particular page at any particular time.

The IRM is grossly unrealistic as a model of locality, as was recognized quite early [?]. It only models hot/cold skew in referencing behavior, and does not model other crucial regularities, such as recency skew.

The IRM assumes that the pattern of references to any particular page is random, but varying around a mean that does not change—the intervals between touches to a page form a random walk around a fixed value.

If real programs were well-modeled by IRM, it would be possible for a replacement policy to adapt quickly to the frequencies of touches to pages and use those to infer the probability of each page being touched at any point in the future. The replacement policy couldn't guess the times exactly, of course, because they're randomized, but it could usually correctly guess the likely range (and probability distribution). For this kind of behavior, LFU would work quite well.

(In fact, it would be the optimal online algorithm, since it exploits all of the information available in the trace—no online algorithm could do better, because there is no other exploitable information in the pattern of past touches to pages.[15])

9.4.2 A simple (first-order) Markov model of reference sequences

Another simple kind of Markov model is a *first-order* Markov model, where the likelihood of a given event explicitly depends only on the preceding event. Each state is characterized solely by what the preceding even was; for each state, a simple weighting of events is used to bias a random function that picks which event will come next.

For example, suppose that we want to model the fact that after touching a particular page, a program is likely to next touch some other particular pages with high probability, but unlikely to touch most other pages. (We might have observed from analysis of a reference trace that touches to page A are usually followed immediately by touches to page P, occasionally by touches to page D, and never by touches to other pages.)

[15] Phalke has proven that LFU is online optimal under IRM assumptions; our take on this is that it only proves that IRM is an extremely bad model of program behavior. LFU is known to be inferior to other simple algorithms, so a proof that it is optimal under IRM is mainly a proof that IRM is wrong.

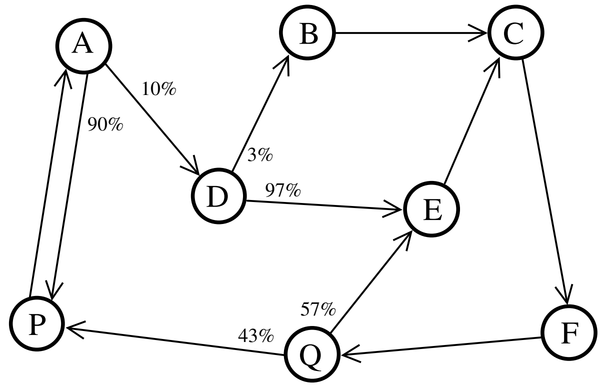

Figure 7: A First-Order Markov Model.

We can model this situation by constructing a graph of states which signify which page was touched last, as shown in Figure 7. The nodes in this graph represent the current state of the program where *by hypothesis* the only interesting fact about a state is what page was just touched. From each node in this graph and to each other node in the graph, we create an edge (directed arc) signifying the possible subsequent states. The edges are decorated with a weight that gives the probability of a transition to that state. For example, the edge between A and P might be weighted 90%, because we have observed that touches to A are followed by touches to P 90% of the time. Likewise, the edge from A to D might be weighted 10%. (In the figure, we have omitted the probabibilities wherever there is only one edge leaving a node; these are implicitly weighted 100%. We have also neglected to specify a starting state, which any Markov model must have.)

If we use statistics in this way to weight the edges between all pages that ever occur consecutively in a trace, we have constructed a first-order Markov of the sequential page accesses.

We can use the resulting Markov model in two ways. One is to use it as a model to generate randomized sequences of page references that reflect these basic statistics. The other is to reason about it formally, and infer what *would* happen (probabilistically) if we generated such a trace.

Generating a reference trace from this Markov model is very easy. We just write a simple loop program which starts in some chosen state, then uses a random number generator to pick between the successor states, probabilistically, using the edge weights from that node in the graph to bias the random selection.

That is, we "simulate" a program that resembles the real program; it keeps a pointer into the directed graph, recording which state it's in—in this case, indicating which page it touched last—and selects among the outgoing edges randomly, but biased by their weights, to determine the next state. The only memory this simulator has of its state is the pointer into the graph, and (implicitly) the connections between the nodes. It doesn't remember which path it actually took to *get* to the current state; all that it "knows" is that it followed some edge that connects to this state.

There are several points to notice about this simple Markov model of reference sequences:

- *It only models extremely short-term regularities.* It only knows the probability that a given state is followed by a given other state, independent of anything that happened previously.

  For example, it only encodes the fact that touches to page A are followed by touches to page P 90% of the time. Other regularities in the access pattern are lost. For example, in the real trace, it may be true that the program goes through alternating phases, where the touches to A are followed by touches to P 900 times in a row, and then touches to A are followed by touches to D 100 times in a row. This fact is lost in constructing the Markov model, and traces generated from the Markov model are extremely unlikely to exhibit this pattern.

- *It is randomized.* The *only* regularities in the Markov model are encoded the edges and the edge weights. At any given moment, the transition to the next state is biased, but otherwise random. This randomness eliminates some regularities (like the phase behavior), but it *introduces others.* The randomization of state transitions may introduce *new regularities.*

  For example, in our hypothetical real program, it may be fairly common for touches to A *not* to be followed by touches to D for extended periods of time (e.g., 900 times in a row). In a trace generated from the Markov model, this is extremely unlikely. A sequence including (say) 100 touches to A will *almost always* contain a touch to D, because each of those touches has a 10% chance of being followed by a touch to D, independently of the others.

Notice that a first-order markov model may capture some regularities in sequences of more than two successive events. The model of Figure 7, for example, captures the fact that a touch to page B is always followed by touches to C F and Q, because those nodes each have only one outgoing edge. At nodes A, D, and Q, however, the choice of transitions is randomized. In the real trace, there may be strong regularities in the sequence of the choices at these points, but the Markov model does not capture them. The paths taken by an actual program may include very distinct patterns over hundreds or thousands of memory references, but these patterns have been *superimposed* in constructing the Markov model, leaving only a weighted random choice at each choice point.

Straightforward low-order Markov models are risky for modeling locality, in that they do not generally capture the essential regularities at the appropriate timescale. Very short-term patterns may be fairly realistic, but larger-scale regularities are often systematically obliterated—such as recency skew at larger timescales relevant to most caches, and multiple working sets that overlap in terms of blocks touched but occur in distinct phases.

Errors in recency skew are likely to affect results for almost any caching policy. Errors due to the mixing of working sets are likely to especially affect results for prefetching and clustering policies.

9.4.3 Higher-order Markov models

One way of increasing the realism of Markov models is to increase the "order" of the model. The states in a first-order Markov model reflect individual events in a sequence, but higher-order models can reflect longer subsequences. Unfortunately, this is often less useful than it seems at first. Whether it actually makes the model "realistic" depends on what the model will be used for; for our purposes, it often doesn't.

A second-order version of our Markov model of reference sequences would have states that reflect the last *two* page references. Rather than having a state for each page that is ever referenced, we'd have a state for each pair of pages that is ever touched in succession. Figure **??** shows a simple second-order Markov model of block referencing.

In our example, rather than having a single state for block A, we'd have a state AP, meaning that we'd

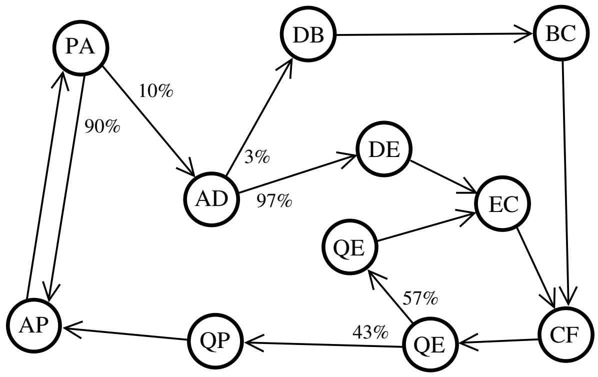

Figure 8: A Second-Order Markov Model.

just touched P, but had touched A before that, and another state AD, meaning that we'd touched block A and then D. The arcs out from AP then indicate which pages are likely to be touched after successive touches to A and P. E.g., if R is touched next 90% of the time, an arc weighted 90% from AP will go to PR. (Recall that each node records the last two touches, but each transition indicates a single touch. A state that ends with P will always be followed by one that begins with P.)

This principle can be carried further, with third-order and fourth-order Markov models, and so on.

Higher-order Markov models often generate sequences that much more closely resemble the original sequences, over a small timescale. Still, however, unless the model is of very high order, large-scale regularities will be lost and new, randomized regularities will appear in their places. Over the long term, the cumulative effect of many randomized choices introduces increasing opportunities for error, with respect to any systematicities in a real program's control structure.

For memory hierarchies, the relevant regularities are typically *not* at the scale of two or five or even a hundred successive memory references, but at the scale of thousands, millions, or even billions of memory references.

Markov models of such high order are quite difficult to deal with in any general way, due to an explosion of states. For a Markov model of order n, we need a state to represent each distinct sequence of n events. (For high-order Markov models, a model "summarizing" a workload could be much larger than the reference trace itself.)

Another problem with high-order Markov models is that they are often mathematically intractable. One appeal of Markov models is that they have certain mathematical properties that support reasoning about what *would* happen *if* the model were constructed. In some cases, it is possible to reason about a Markov model without actually building an explicit representation of the model itself (i.e., the directed graph). Unfortunately, such reasoning often has two weaknesses:

- *It often requires auxiliary assumptions which may be false.* Considerable work has been done on *ergodic* Markov models, which are well-behaved in in a certain mathematical sense [**that will be described later**].
- *Such models are extremely difficult to validate.* Because the computations over high-order Markov models are often intractable, it is often difficult to validate the assumptions required to ensure that realistic models are in fact analytically tractable.

9.4.4 The Independent Reference Interval Model—a low-order Markov model of higher-level regularities

Another approach to using Markov models is to avoid the modeling of concrete program events—such as touches to particular blocks—and model something more abstract, and more relevant to the problem at hand. We refer to these more abstract properties as "higher level" regularities, but that is a separate issue from the "order" of a "higher order" Markov model.

For example, if we are interested in the behaviors of a program that are relevant to LRU caching, we may abstract away from *which* blocks are touched, and model only the relevant facts about *how* blocks are touched. For LRU, what we generally care about is how often we touch blocks soon after the last touch, and how often touches to blocks are separated by touches to many other blocks.

The relevant statistics for this are the frequency of touches to *recency queue positions*—concretely, an LRU distance histogram—rather than the frequency of touches to the particular pages at those positions.

The *Independent Reference Interval Model* (IRIM) is a zeroth-order Markov model of memory referencing based solely on recency distributions. It has a single state, connected to itself by arcs that reflect the frequencies of touches to (blocks at) different recency queue positions.

To generate a trace from such a model, we can loop to generate successive references, as with the IRM. To initialize the system, we can assign each page a position in the recency queue.[16] At each iteration, we use the arc weights to bias a random choice among the edges, and emit a reference to whatever page is at the corresponding recency queue position. This will generate a trace with the recency locality statistics that were used to construct the Markov model.

[To do this, we must add some machinery that IRM doesn't have—we have to keep track of the contents of the LRU queue, so that references to recency queue positions can be translated into references to particular pages. This is still a Markov model, because the overall process is driven by a *Markov process*—the finite state machine)—but there is other state information too...]

Notice that in constructing an IRIM model, all regularities of the original program have been systematically discarded, except for those that are relevant to purely recency-based caching policies such as LRU. The original page numbers have been forgotten, and the model encodes *exactly* and *only* the recency distribution of the original trace—all of the other information in the resulting trace is random noise, supplied by the Markov model.

Any attempt to measure the performance of a non-LRU policy using a synthetic IRIM trace is likely to contain serious errors. Any pattern in the original trace that the policy might have adapted to has been eliminated, and the policy will only do well if it exploits recency skew and deals well with random noise.

At this point, you may be wondering *what use is it?*, and this is certainly an excellent question. At first glance, there doesn't seem to be anything that the IRIM model is good for that an LRU distance histogram doesn't do at least as well. For analyzing LRU memories, an LRU histogram can give exact miss rates with no need for a Markov model, or a synthetic trace, or actual simulation runs. For many non-LRU memories, *the relevant information has been discarded*—a non-LRU policy has a greatly reduced chance of beating LRU, because an IRIM trace contains *exactly* and

[16] How we do this doesn't usually matter much. We can also start with an empty queue, indicating that the cache is empty at the start of the trace, and assign pages to queue positions lazily.

only the information relevant to LRU-like replacement decisions.[17]

The main utility of the IRIM model is in providing a very simple and tractable model of locality, which can be compared against actual experimental results in enlightening ways.

For example, a real trace and an IRIM trace derived from it may both be run through a simulator for a proposed cache. If the two results are very close, this suggests that the *most important* regularities in the trace, *for the purposes of that cache design*, are those captured by the IRIM model.[18] If the results are *not* close, the existence of errors in the IRIM predictions suggest that other interactions between the cache and real program behavior are significant; the pattern of errors may give a clue as to which regularities are important.

9.4.5 Fundamental limitations of Markov models

So far, we have discussed two limitations of Markov models:

- *They tend to capture only short-term regularities in sequences*, unless the model is of an extremely high (and likely impractical) order, and
- *Using higher-order properties discards other properties that may be relevant*, as when the IRIM model discards information about *which* blocks are referenced in which order.

There is a fourth limitation, however: Markov models only capture certain simple *kinds* of regularities. *Real programs often exhibit strong regularities which are not captured by any reasonable Markov model, of any order.* A Markov model is only suitable if the regularities it doesn't capture are *known not to be relevant* to the purpose for which the model is used.

A Markov model is a finite automaton, like regular expression, and a Markov model shares many of the limitations of a regular expressions in capturing interesting regularities. It has a fixed set of states, and no memory of the transitions that led to a particular state. As such, it is extremely unlikely to generate certain kinds of simple and common patterns which can be generated quite easily with other simple machines.

Regular expressions cannot be used to characterize many important kinds of patterns, the best known being nested expressions—for that, a context-free grammar (at least) is required—a stack memory is needed, to keep track of the level of nesting, and to notice when the ends of expressions are matched. For more sophisticated patterns, still more sophisticated machines are necessary, such as transformational grammars or machines that aren't easily described in grammatical terms at all [Cho56].

A Markov model is used to *generate* patterns, but the same limitation applies. Because it is randomized (a nondeterministic automaton), a Markov model *may* generate almost any kind of sequence—within the possibilities defined by the edges in the graph—but is probabilistically unlikely to generate large, interesting patterns that do not "look Markov."[19]

Regular expressions capture *mostly literal sequences of tokens*, entirely missing any patterns due to nesting or other not-strictly-sequential kinds of patterns. (They can capture certain kinds of repetition of literal sequences, but are otherwise quite limited.)

Likewise, a Markov model is overwhelmingly biased toward certain kinds of randomized "approximately literal" sequences, or mushings together of sequences. There is variation, due to randomization, but the variation is unstructured and intrinsically irregular.

Consider a Markov model of concrete sequences of memory accesses. Given a model of sufficiently high order, or sufficient skew in the edge weights, certain literal sequences are likely to occur repeatedly. However, these realistic subsequences are will be linked together in randomized ways to form the overall event sequence.

Consider two patterns of memory access which are plausible (and, we think, common):

- Touching the same blocks in the same order repeatedly;

[17] This is not strictly true, because a non-LRU policy might be able to read the recency distribution better than LRU, and "beat it at its own game." Opportunities for beating LRU are greatly reduced, however, because most regularities from the original trace have been systematically eliminated. This includes lower-level regularities such as particular sequences of page accesses and higher-level regularities such as systematic changes in the recency distribution due to phase behavior.

[18] Of course, this doesn't *prove* that no other regularities in the original trace are relevant. There could be other regularities whose effects tend to cancel each other out.

[19] It is rather like 600 monkeys typing away. Even with a fairly high-order Markov model of the word sequence Shakespeare's texts, they are extremely unlikely to generate *Twelfth Night*—or anything with a remotely reasonable and sustained plot structure—in millions of years. In the meantime, they may generate quite a few new and very Shakespearean phrases or even sentences, but that's beside the point.

- Touching blocks first in one order, and then in the *opposite* order.

The first pattern is characteristic of many loops. A Markov model may capture this pattern reasonably well, especially if this is the *only* pattern of references to any of those blocks. In that case, the edges connecting the states may directly reflect the sequential ordering, and the lack of edges to other states may ensure that once one of the blocks is touched, the succeeding blocks are touched in the right order.

However, if any of those blocks is ever touched in any other way, the presence of arcs to other states is likely to derail the Markov model, so that it seldom completes a whole loop through the set of pages. Where the original program may have characteristically looped through the whole set *or* done something quite different, the Markov model of the program will tend to loop through *part* of the set and get "derailed" by other edges.

In terms of timescale relativity, this may have a major effect on locality. Where each of the real paths through the page graph has definite implications for locality, e.g., touching a certain number of pages at each iteration of a loop, a random walk through the superimposed paths (Markov model) may be quite different.

The second pattern appears not to be uncommon, either. Some programs construct data structures in one order, then traverse them in the opposite order. (Perhaps because that's the chosen order of processing, as in many algorithms using a stack, or perhaps just to free their storage.) This pattern may also be imposed by memory allocators that tend to reuse memory in roughly LIFO order.

Here a first-order Markov model may do extremely poorly, because the edges in the graph go both directions—the direction of the first traversal of the data, and the direction of the opposite traversal. Where the original program went all the way through the data in one direction, and all the way through in the other, a first-order Markov model will tend to wander around, going one way and then the other, because the arcs are weighted equally.

This clearly has strong implications for locality of reference. Where the original program cruised through the entire set quickly, twice, the Markov process will tend to wander aimlessly back and forth over a few items, reversing direction half the time, and just drifting slowly one way or the other. *A random walk over a Markov model can have very good locality, in a way that does not reflect the locality of the real patterns that were superimposed to generate the Markov model.*

Again, the presence of extraneous edges due to other patterns of access to the same data may derail the Markov process. It may tend to mostly wander back and forth over a few items until it randomly selects another edge and goes off and does something entirely different.

A higher-order Markov model may do better here, because it will have states that implicitly represent the directions of the traversal. (For example, a second-order model will be in state AB after touching block B going one direction, but in state BA after the opposite traversal.) This may result in a tendency to traverse the entire set of blocks in one direction sometimes, and the other direction sometimes, with less chance of getting derailed; other edges into and out of the set may be ignored because they do not enter the set "in the right way," with the appropriate sequence of two block touches.

Even with a higher-order Markov model, we are definitely not out of the woods in terms of simulating realistic program behavior, even for simple LRU caching. We *might* get loop-like or oscillating patterns that resemble patterns in the original program, but there are many opportunities for the Markov model to be unrealistic.

9.4.6 Problems with phase behavior

Many programs exhibit strong phase behavior, and this is problematic for straighforward Markov modeling techniques. Markov models are based on statistical weighting of individual events, or short sequential patterns of events. Phase behavior is something else.

If we model concrete memory referencing straightforwardly using Markov models, we will lose information about the time-varying nature of block accesses. For example, a certain region of memory may be allocated to hold certain data structures during one phase of a program, then later deallocated, and then reused to hold entirely different data structures during a later phase. The resulting Markov state machine may include edges for different phases, superimposed.

If the access patterns are very different at a small scale, then a higher-order Markov model may manage to distinguish between them, but if some pages are accessed in the same sequence in different phases of program execution, the parts of the state graph that represent different phases' behavior will become connected. The resulting Markov process will tend to probabilistically switch between the behaviors characteristic of one phase and those characteristic of another phase, when it encounters states that are shared between two phases. This will tend to "mush together" aspects of different kinds of phases, and the result may be quite different from the original phase behavior of the program being modeled.

For example, if one (real) phase accesses a fairly small amount of data, and another accesses a large amount of data, the locality characteristics of the latter may dominate the program's locality characteristics. (The "small" phase may incur very few misses, while the "large" phase incurs very many.)

Mushing the two kinds of phases together in the overall (superimposed) state graph may tend to make a single kind of phase that accesses an intermediate amount of data. For a small cache, this may be worse than having two very different kinds of phases, because it the cache may be too small to do well for *any* phase. For a large one, it may be better than the original program, because all simulated phases fit in the cache. On the other hand, mushing the two phases' state graphs together may do something else entirely, depending on how the graphs happen to end up connected. It might be that all phases tend to traverse all of the data, making locality uniformly worse.

It is reasonable to wonder how high-order a Markov model must be to keep the resulting state graph from becoming "too connected." Since constructing the Markov model effectively connects potential paths at shared states, reducing the number of states that are shared across (real) program phases should make it "more realistic": a higher-order Markov model will have fewer shared states, since distinct paths will tend to have distinct states (encoding more consecutive touches), and fewer spurious connections at nodes connecting shared (short) subsequences.

Unfortunately, the answer to this question is not reassuring. For a spurious connection to be created in a Markov model of order n, it is only necessary for distinct phases to touch the same n pages in the same order. This does not seem unlikely for any small n—all that is necessary is for both of the distinct phases to execute some routine that touches the same n blocks in the same order. For example, two phases might both call a routine that traverses a shared linked list that crosses block boundaries n times.[20]

9.4.7 Ergodicity

Some theoretical papers on the modeling of program behavior assume that programs can be modeled by ergodic Markov models, i.e., Markov models whose behavior is stable in the long run in a convenient mathematical sense.

In an ergodic Markov model, the chances of being in any particular state are fixed in the long run. That is, if the Markov process is allowed to run for a long enough period, it has a nonzero chance of visiting each state, and over an even longer period, it will tend to visit the different states in definite proportions.[21]

Sometimes, this assumption is clearly stated as a convenience (in effect, "we hope they're ergodic, because if they're not, we can't do the math"), but in other cases a stronger claim is made about the realism of this approach—it is strongly implied (if not stated) that the models are in fact realistic.

Ergodic Markov models are "stable" in the sense that their long-run behavior tends to show similar distributions of visits to states within a sufficiently large time window. Unfortunately, there are two problems with ergodic Markov models for modeling real program behavior:

[20] The simplest example of this is a list whose elements are all in distinct blocks, but a list that crosses back and forth across a few blocks will trigger the same problem. Similar examples arise for tree traversals and numerous other operations.

[21] An example of a non-ergodic Markov model is one in which some reachable subgraph has no out edges—once the Markov process enters that subgraph, it is "trapped" and cannot escape to any other parts of the graph. A model with two or more such subgraphs may be very unpredictable, in the sense that its long-run behavior depends on which subgraph it happens to wander into first.

At this point, we should admit that we have oversimplified our discussion of Markov models somewhat, because there are other kinds of Markov models that may not be ergodic.

In general, a *Markov process* is characterized by a finite state machine, but a *Markov model* may have some auxiliary state, "driven" by the Markov process, as in the case of IRIM's recency queue.

For example, in memory allocation studies, the order of a program's memory allocation and deallocation requests is often assumed to be describable by a Markov *process*, but the state of the heap memory is included in the overall *model*. (In general, even the former assumption appears to be systematically false, for several reasons [WJNB95].)

Such a model may not be ergodic, even if the Markov process itself is ergodic. For example, if a Markov process generates allocation requests in proportions that are not balanced by corresponding deallocation requests, the memory requirements may increase without bounds. Thus, the Markov process may be stable in the long run, generating probabilistically definite proportions of requests of different types, but the Markov model may not be, because the resulting heap memory usage is unstable.

- *They may be unrealistically stable*, because some real programs are not stable in the appropriate sense, and

- *their stability may be of an unrealistic type.*

Many programs are simply not stable in anything like the sense of an ergodic Markov model, at least where memory referencing is concerned. They have striking phase behavior that differs in each phase in several respects, even if the phases represent the same operation over different data sets. (Consider a compiler, compiling a file of procedures, each of which is typically different in size and nesting of constructs.) The relevant regularities may not be regularities in the sequence of accesses to particular memory blocks, or even in strict sequences of accesses to logical (program-level) data objects.

Even for programs whose behavior is unusually "stable," an ergodic Markov model is likely to be a poor model. A real program may exhibit certain kinds of stability that resemble an ergodic Markov model, such that *some* ergodic Markov model *could* be constructed and give realistic results for a certain class of uses. For example, a program might happen to be structured in such a way that over the long run, it tends to touch each block with a roughly fixed frequency, and touch things in the same short-term sequential order in fixed proportions.

Even if this is true, however, this does *not* necessarily mean that a straight Markov modeling of the program will yield a realistic model. If we simply use the relative frequencies of state transitions from each state to build a weighted graph, the resulting Markov model, being randomized, may tend to fall into a certain stable kind of random walk, but that stability may not reflect the stability of the real program—the real program may be stable for different reasons, having to do with its high-level control structure.

For example, consider a program with alternating phases, one of which loops through a set of blocks several times in one direction, and another which loops through the same data several times in the opposite order. This program's behavior is stable in the long run, in the sense that it touches all of the looped-over data the same number of times in the same directions at each repetition of the pair of phases. This does *not* mean that at any given moment, it is equally likely to go either direction through the data.[22]

The deep problem here is that Markov modeling is fundamentally syntactic, and based on a fundamentally impoverished kind of syntax.

9.4.8 General comments on Markov models

In general, real programs do not behave like Markov models. They simply are not stochastic processes driven by random variables—they are executable plans, and plans have structure that is often complex, but usually not random. This structure is often not mathematically simple, and randomness only confuses the issue. Complex, structured plans often do in fact exhibit strong and simple regularities, but these regularities are often *not* of the sort that Markov models capture.

"Unknown" behavior cannot safely be modeled as random, for two reasons:

- *Unknown behavior may be patterned in important ways that affect locality.* Replacing real behavior with randomness eliminates systematicities that may affect different caching policies differently.

- *Random behavior itself has important consequences.* Randomness is not "neutral" with respect to locality. Randomization can decrease locality (e.g., by randomizing which pages are touched) or increase it (e.g., by causing random walks over a narrow range of items).

Given the foregoing, we believe that Markov modeling is a *prima facie* suspect approach to the modeling of program behavior. Of course, this does not mean that Markov models are entirely useless—just that they must be used with extreme care.

Markov modeling based on higher-order properties (such as recency skew, or other logical properties of programs) may be appropriate in many cases. For example, the IRIM model works fine for a limited class of applications where the main relevant regularities are expressible in terms of recency skew.

In general, *any* model is limited by how well it captures the relevant regularities for a particular kind of purpose, and relevance depends on the purpose.

The use of Markov models, though very widespread, is far from a panacea. [**You can't just put a bunch of data into a mathematical formalism and turn**

[22] It will always loop, and never walk randomly back and forth, as a zeroth-order Markov model might. It will always loop a fixed number of times in each direction, as even high-order Markov models generally won't.

a crank to get a scientific model. Why do people keep thinking you can? Just because everybody seems to do it?] In many cases, Markov models are mathematically tractable but scientifically ill-considered. The main work in modeling must always be *understanding the phenomena to be modeled*, and constructing a model that captures the *relevant* features of the phenomena for a particular purpose. Program behavior is very far from being so well understood as to yield simple, general, and predictive mathematical models.

Markov models fail to directly capture many of the obvious regularities in real program behavior, because real programs are not finite-state machines in any useful sense. Markov models can still be applicable in some cases, but applying them appropriately requires understanding the phenomena well enough to cast the Markov model in a reasonable way, e.g., by deciding what relevant features of a program state should count as a state in the Markov model, and whether in fact the regularities in the state sequences are reasonably modeled by stochastic processes at all.

Since the underlying processes—real computer programs—are clearly not stochastic processes in general, the burden of proof is on the modeler to show that the weaknesses of stochastic models do not matter. It is necessary to explain why the non-stochastic properties of real programs don't matter, and why a randomized model is "close enough."

9.5 Modeling Fully Associative Caches

9.6 Modeling Virtual Memories and Multiprogramming

9.6.1 The Working Set Model

9.6.2 Page Fault Frequency.

9.7 Modeling Effects of Associativity

9.8 Modeling Effects of Context Switching

9.9 Modeling Instruction Streams

9.9.1 The loop model

9.10 Models for Clustering

9.11 Hifalutin' Models

[I don't actually know what to call this section... it's about models that are mathematically interesting, but usually based on very dubious overgeneralizations, and premature mathematization of things that are just *not* mathematically well-behaved.]

9.11.1 Fourier Models

[These make a lot more sense than most highly mathematical conceptualizations of locality, but they're not quite right. Basic idea is that you separate out the high-, middle-, and low-frequency behavior, which is a nice step toward timescale relativity. The problem is that the Fourier transform isn't quite the right one, because it doesn't take into account the interactions between superimposed behaviors, and "frequency" in a harmonic sense is not usually what's crucial. LRU transform is better, because it's more directly related to caching considerations. Gap model is also better, because it's directly related to idleness.]

9.11.2 Fractal Models

Basic claim is that program behavior is fractal, because miss rate curves or inter-miss gaps often follow a particular kind of smooth, self-similar distribution.

In a technical sense, this may often be true, if you look at a summary over a whole run of a program, but it misses the structure of the trace. It also misses the point that program behavior is generally *not* fractal in any *interesting* sense. Some side-effects of program behavior may look fractal if viewed through the wrong end of a telescope, but if you're going to blur things that way, you're better of characterizing real tradeoff curves rather than forcing them into a trivializing mathematical model.

Some of these papers are thoroughly bogus—claims about hyperbolic curves when the data visibly don't fit those curves, due to real phase behavior. Never well validated—usually one or three traces.

If these models are taken as describing cache effectiveness, they don't seem to offer anything beyond rather bad curve fitting. If they're taken as describing program behavior in any deep sense, they're basically just wrong.

10 Empirical Methodology

[blah blah... need to move some stuff from beginning of previous section to here, now that I've split analytic and empirical into separate sections.]

This simulation can be quite expensive in terms of storage costs for real reference traces (the input data to simulators), and in main memory and especially CPU time. Many hundreds of experiments may be run for different combinations of several design features and parameters, and each of these simulations may run for hours or days.

[empirical stuff is preferred...]

The high cost of extensive simulation using real traces has motivated the development of several techniques to reduce the costs of tracing and simulation. These fall into several categories: tools for gathering detailed traces conveniently, data reduction tools for reducing the size of the stored traces without appreciably affecting the results of simulations using them, and efficient simulation algorithms.

10.1 Synthetic Benchmarks

Simulation experiments sometimes are done using *synthetic benchmarks* in lieu of real data gathered by tracing real programs. Simple programs are constructed so that they exhibit behaviors thought to resemble real application program behavior; these synthetic programs are then used in experiments. *Such results must be interpreted with extreme caution.* Often, these programs reflect their designers' intuitions about program behavior, and these intuitions may be quite wrong.

Worse, the synthetic programs often implicitly incorporate unrealistic assumptions underlying common analytic models. The apparently "empirical" nature of these "experimental" results often lulls people into thinking that more is known about real programs than actually is known. This is particularly common when synthetic programs are applied beyond the scope for which they were originally designed.

A benchmark may resemble real programs in certain ways that are relevant to certain aspects of system design, but in other ways, synthetic programs generally say less than nothing about the behavior of real programs. In any case, benchmark performance must be validated against the behavior of real programs before experimental results can be taken very seriously. In general, little validation of this sort is done, so many experimental results are extremely dubious. Even when a benchmark has been validated with respect to certain issues, it is still suspect for any purpose for which it has not been validated—in general, simple synthetic programs do not exhibit the same kinds of irregularities and irregularities that are relevant to interesting memory hierarchies. Like analytic models, the results of using them for novel purposes are likely to be systematically wrong. (As we will explain later, many synthetic programs actually approximate first-order Markov models—though this is often not recognized—and exhibit many of the same potential errors that the corresponding analytic Markov models do.)

This is not to say that synthetic programs are never useful. Synthetic programs often have the advantage that they can be varied systematically, by changing parameters, and this allows experimentation with a wide range of possible "program" behaviors. Such results should be interpreted very cautiously, but can be quite informative, by indicating which features of program behavior interact with which aspects of memory hierarchies. This can give interesting insight into the tradeoffs involved in system design, and point out important aspects of real program behavior that must be studied.

10.1.1 General Issues in Benchmarking

10.1.2 Problems with Synthetic Data

10.1.3 Synthetic Benchmarks and Clustering

Synthetic benchmarks have been used in most studies of object clustering for object-oriented databases. In part, the use of synthetic benchmarks is a convenience; large data sets for database systems are hard to obtain and adapt to a novel database system. This is often especially hard in object-oriented database systems, because the technology of object databases is rapidly evolving, and systems are often incompatible.

[blah blah blah]

Most object database benchmarks use a few simple data structures, but interconnect large numbers of small objects in a random or semi-random manner.

The algorithms in these benchmarks are fairly simple, and typically follow pointer many pointer links in simple ways, e.g., using a breadth-first search of objects reachable from some randomly-chosen starting object.

In effect, the randomized interconnections between objects are similar to a Markov model, and tend to bias simulated "program" behavior toward simple stochastic behaviors. The behavior of a simple algorithm is primarily determined by its blind traversal of a randomized graph.

Because the locality properties of the programs are primarily determined by the static connections between the items in the database, rather than any interesting control structure in the program itself, assumptions about stochastic behavior are likely to be true for these programs *even if they are not true for real programs*.

Equally important, the locality properties of the programs operating on the data are likely to be extremely strongly related to the static "locality" in the ways that objects are created—if a pointer from one object to another exist, a blind traversal generally *will* traverse that pointer.

The odds of touching an object therefore depend almost entirely on the stochastically distributed connections to it—there is very little phase behavior that is not directly correlated to the shape of the graph of connections among objects.

Several experiments have been performed to measure the effectiveness of various object clustering strategies for these synthetic benchmarks, without validation against real programs. The results of these experiments should be interpreted very cautiously.

OO1. In the OO1 (Object Operations One) benchmark, the database consists primarily of a single indexed collection of *part* objects, plus connections between the parts. The part objects are indexed by numeric keys, and the connections between the parts are correlated with their key values.

Each part is connected via a directed link (pointer) to each of three other parts, chosen partly randomly. 90% of the parts are to "nearby" parts, i.e., between objects whose key values are within 1% of each other. The other 10%, however, connect to (uniformly) randomly chosen parts.

The first thing to notice about this randomized interconnection scheme is that while it exhibits some locality—90% of connections are local—it actually has disastrous effects on the locality of simple algorithms operating over the data. On average, *every tenth link traversal will access a randomly-chosen part object.* Because of this, OO1 has extraordinarily poor locality of reference.

(The OO1 designers were aware of this, at least to some degree; they specify that traversals be executed for both "cold" (empty) caches and for "hot" caches, which already contain the data to be traversed. Thus OO1 provides two extremes of behavior—very bad locality, and very good locality—which can be used to roughly assess a system's performance under two very different kinds of use. Unfortunately, there is little information to guide the interpretation of the results; it is unclear what an "expected" mix of these kinds of behavior is, and they often tend to get equal weight. We believe that results from OO1 have often been misinterpreted in assessing the relative merits of systems.)

This unusually poor locality may not matter for some purposes, but for others it is crucial. For example, some object databases incur overhead at every pointer operation, while others incur overhead primarily at page faults. In general, the "fine-grained" systems incur several instructions overhead at each pointer traversal (and perhaps each pointer comparison), while the "coarse grained" ones may incur thousands of instructions of overhead at a page fault. If the frequency of pointer traversals is several orders of magnitude higher than the frequency of page faults, the coarse-grained techniques are more efficient. For normal programs, this is almost always true. (Recall that on a modern computer, a program that takes a page fault every million instructions is probably paging heavily).

For object databases, this is less clear[23], but it is clear that OO1's lack of locality raises serious questions. It seems to exhibit two extremes of behavior, but **[little middle-ground behavior...]**

OO1 has been used for experiments in clustering, but the structure of the benchmark itself tends to favor certain clustering strategies over others. OO1 seems to be unrealistically Markov-like in two ways:

- The important links are of a single type (from part to part) and tend to be traversed in a uniform way—all outgoing links from a part are usually traversed. The major variation is just in the weighted (90% nearby) randomness of the links. There are no important *kinds* of data objects that are traversed during some operations but not others.

  This stresses the first half of the clustering problem (keeping together things that are accessed together) at the expense of the second half (keeping *apart* things that are accessed *differently*).

[23] **[blah blah... databases tend to be I/O bound, but OODB's are especially likely to be used for more CPU-bound tasks like CAD... cite Tiwary]**

- The only useful information in the graph of part objects is the connectivity of the graph. The skew in the heat of the parts and links is due to the presence of densely-connected sets of parts, which arise from the biased random distribution of links.

  This ensures that there will be a strong correlation between the static structure of the graph and the dynamic locality of the (simple) traversals of the graph.

  Further, it means that locality characteristics are likely to be *consistent*, especially with respect to the relative heat of links. If an object or link is hot during one traversal, it is very likely to be hot during any other traversal that encounters it at all.

The structure of OO1 lends itself to certain kinds of clustering techniques, and not others, in ways that may not be realistic.

- Because all phases of program execution are similar (e.g., a series of breadth-first traversals from random parts), the skew in object and pointer heat is likely to be fairly consistent across operations.[24]

  This may favor simple connectivity- or heat-based schemes over schemes which use heuristics based on object type, because there is not much useful type information to exploit.

- The fact that all operations are data-intensive rather than compute-intensive may make the edge weights an unrealistically close approximation of actual importance for caching. (In a real program, some edges or objects may be very hot in phases where locality is excellent, but still not particularly important for clustering. Cooler edges traversed at widely-spaced times may be more important.)

  This may favor simple heat-based schemes relative to either simple connectivity-based schemes

More generally, the fact that the benchmark is so simple ensures that only certain things matter to clustering; since the other characteristics of the benchmark are random, there is little opportunity for sophisticated clustering schemes to exploit the same regularities that they might exploit successfully for real programs.

[24] E.g., breadth-first traversals starting at different parts, but encountering the same densely-connected subgraph, or a single traversal that encounters the same densely-connected subgraph multiple times.

007.

LabBase.

10.1.4 Fundamental Problems in Designing Benchmarks

10.2 Trace-driven Simulation

10.2.1 Memory-level Traces vs. Object-level Traces

10.2.2 Gathering Traces

10.2.3 Efficient Simulation for Inclusion-preserving ("Stack") Algorithms

References

[BC92] Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer Verlag.

[BL92] Thomas Ball and James Larus. Optimal profiling and tracing of programs. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 59–70. ACM Press, January 1992.

[Bla83] Ricki Blau. Paging on an object-oriented personal computer for Smalltalk. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Minneapolis, Minnesota, August 1983. Also available as Technical Report UCB/CSD 83/125, University of California at Berkeley, Computer Science Division (EECS), August 1983.

[BS76] Jean-Loup Baer and Gary R. Sager. Dynamic improvement of locality in virtual memory systems. *IEEE Transactions on Software Engineering*, SE-2(1):54–62, March 1976.

[Cho56] Noam Chomsky. Three models for the description of language. *I.R.E. Transactions on Information Theory*, IT-2, September 1956.

[CK93] Robert Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. Technical Report UWCSE 93-06-06, Dept. of Computer Science and Engineering, University of Washington, Seattle, Washington, 1993.

[Cou88] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[Fot] J. A. Fotheringham. Dynamic storage allocation in the atlas computer including an automatic use of the backing store. *Communications of the ACM*, 4:435.

[LWM92] Michael S. Lam, Paul R. Wilson, and Thomas G. Moher. Object type directed garbage collection to improve locality. In Bekkers and Cohen [BC92], pages 404–425.

[Quo94] R. W. Quong. Expected i-cache miss rates via the gap model. pages 372–383, April 1994.

[RO91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, pages 1–15, Pacific Grove, California, October 1991. ACM Press. Published as *Operating Systems Review 25*(5).

[SKW92] Vivek Singhal, Sheetal V. Kakkad, and Paul R. Wilson. Texas: an efficient, portable persistent store. In Antonio Albano and Ron Morrison, editors, *Fifth International Workshop on Persistent Object Systems*, pages 11–33, San Miniato, Italy, September 1992. Springer Verlag.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3), 1985.

[VC90] P. Vongsathorn and S. D. Carson. A system for adaptive disk rearrangement. *Software Practice and Experience*, 20(3):225–242, March 1990.

[Whi80] Jon L. White. Address/memory management for a gigantic Lisp environment, or, GC considered harmful. In *LISP Conference*, pages 119–127, Redwood Estates, California, August 1980.

[Wil] Paul R. Wilson. Garbage collection. *Computing Surveys*. Expanded version of [Wil92]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps`. In revision, to appear.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [BC92], pages 1–42.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, December 1993. Expanded version of workshop position paper submitted for publication.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–191, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992.