ON THE MATHEMATICS OF CACHING

By

Mark W. Brehob

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science and Engineering

2003

ABSTRACT

ON THE MATHEMATICS OF CACHING

By

Mark W. Brehob

Computers circuits get faster at a much more more rapid rate than memory. In order for historical computer performance gains to continue, memory latency and bandwidth need to continue to improve. The most common way provide this performance is to use small, but very fast, memory devices to store the most commonly used data called *caches*. Cache performance relies upon *locality*; the predictable way in which caches tend to access memory elements that have accessed in the past or near those previous accesses.

Of fundamental importance to improving a complex device is understanding how that device functions. This work contributes to that understanding in two ways. First, it expands upon an old and mostly unused model of locality called 'stack distance' and shows how that locality interacts with various types of caches including victim and skew caches. Second, a negative result is provided showing that it is computationally intractable to find an optimal schedule for most non-standard caches. This result means that a very useful tool for cache evaluation is not available when working with these non-standard caches. It also means that those working on finding a tractable optimal algorithm for these caches can stop—they will not succeed.

Dedicated to my daughter, Kira Nerys Brehob, and her mother, Katrina Marie Brehob.

Acknowledgments

This dissertation has been many years in the coming and many many people have been more supportive than I could possibly deserve. A short list of those people include Suresh Chittor, Wen-Hann Wang, and Nick Wade of Intel as well as Eric Torng, Stephen Wagner, Uthaisombut Patchrawat, Barbara Bircher Gannod, Travis Doom, Jennifer White Doom, Hugh Smith, and Charles Severance of Michigan State University. My in-laws, parents and sister have stood behind me the whole way. My adviser, Richard Enbody has been remarkable in his support and encouragement. My wife has sacrificed her time and energy beyond anything I could ever repay.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and overview

Modern society has come to rely upon the computer and its ever-increasing performance gains. In order to meet future expectations, both of society in general as well as the computer industry, faster and better computers need to be designed and built. The computer industry, citing a variant of Moore's Law [39], has traditionally expected that processor performance will improve by 60% per year (this is equivelent to the more commonly stated "double in performance every 18 months" Of course, there are many roadblocks on the path to continuing at this rate of exponential performance improvement. These roadblocks include physical limitations of the devices being built, the difficulty of designing extremely large and complex processors, and the amount of heat which must be removed from the processor.

One of the more immediate roadblocks is the relatively slow rate of improvement in the speed of memory devices such as DRAM. This looming "memory wall" [80] stands squarely in the path to continued processor improvement. DRAM process improvements have historically generated a 7% improvement in latency per year. One professor at Berkeley jokingly contrasted this much slower rate of improvement with Moore's law by calling it "Less' Law. [22]"

The most common way to address this disparity is to use small, but very fast, memory devices to store the most commonly used data. Those devices are called *caches.* Of course, not all of the needed data can be kept in these small caches, so some of the memory accesses must go to the DRAM. Those DRAM accesses are said to have been cache misses.

Traditional methods for caching data are reasonably effective, but the widening gap between memory performance and the expected processor performance has made further decreasing the number of cache misses an important goal. As such, many novel caching techniques have been proposed in the last ten years. One difficulty with this plethora of ideas is that they are very difficult to evaluate. Certainly one can compare them in an environment where the workload is held fixed and "similar" sized caches are compared. (In fact we have done such a study for a number of these techniques [14]). However, such studies give very little insight into why these caches perform well—something very important to anyone who proposes to use them in a commercial processor. The focus of this dissertation is to examine what can and cannot be done to gain that needed insight.

The work in this dissertation provides some insight and results about how a cache interacts with the pattern of memory requests made by the processor. The two main contributions are:

- An old technique, LRU stack distance, is greatly expanded upon. It is used to evaluate locality and understand how locality and caches interact.

- An important evaluation technique, the optimal replacement algorithm, is shown to be NP-hard to compute for a large and important class of caches. This means that this optimal replacement is a tool that designers of those caches, which include the "newest" cache designs, will not be able to utilize.

1.1 Overview

As with most scientific endeavors, this work would be meaningless without the context of other's efforts. As such, Chapters 2 and 3 summarize that context. In Chapter 2 a short tutorial on caching is presented along with a discussion concerning a number of non-standard caching techniques. The tutorial mostly consists of material one would find in a standard undergraduate class on the topic. However, the non-standard cache discussion should be novel to most readers.

Chapter 3 is a discussion of the memory reference stream of a processor. These patterns (called locality of reference) are what allow small caches to perform as well as they do. The various techniques in the literature for quantifying and taking advantage of the nature and patterns of these reference streams is examined. At the end of that chapter the stack distance model, which is used throughout Chapter 4, is introduced.

While bits and pieces of original work are scattered thoughout the early chapters, it is Chapters 4 and 5 that contain the original work found in this thesis. Chapter 4 discusses how the stack distance model can be used to visualize and measure the interaction of locality and caches. Specifically that model is validated more extensively than before, new applications are found and the model is expanded to work with non-standard caches.

Chapter 5 is a formal proof, showing that it is NP-hard to compute the optimal replacement policy of certain non-standard caches. This result holds for skew caches, generalized victim and assist caches, and in fact nearly all multi-lateral cache schemes. Further, various results are given which show that it is also NP-hard to closely estimate the number of misses an optimal replacement algorithm would cause on these caches.

1.2 On Terminology used in this document

In the computer profession, as in most professions, terminology is not always as standardized as its practitioners might like. In some cases this extends beyond terminology and into differing standards and assumptions. Below is an attempt to address those terms that might be unclear.

- Throughout this dissertation, the function $lg(x)$ will be used in place of $log_2(x)$ as is common in many Computer Science texts.
- A consistant big endian [39] notation shall be used thoughout this work.
- The SPEC CPU2000 benchmark suite[21] is refered to thoughout this document as the 'SPEC' benchmark suite. The integer and floating-point parts of the benchmark are refered to as SPECint and SPECfp.

Chapter 2

On Caching and Non-standard Caches

This chapter serves two purposes and as such is broken into two sections. The purpose of the first section is to introduce the fundamentals of processor caching. This section does not, and cannot, provide a complete overview of the topic. Rather, its purpose is to provide a basic understanding of terms that will be used in further chapters. The experienced computer architect will find the vast majority of the material in that first section to be review, while others may find the need to refer to computer architecture books such as [39, 56, 66] in order to provide needed context.

The second section of this chapter focuses on non-standard caching schemes. This material is significantly more obscure than the material of the first section[1]. Because Chapters 4 and 5 assume a working knowledge of this material it is also included in this chapter along with the more basic material of section 2.1.

[1]In fact, only one paper has been found which provides an overview of even a few of these caching schemes [82].

2.1 Introduction to Processor Caching

As a processor executes instructions it needs information from the memory system where both data and the instructions themselves are stored. However, the main memory is usually very slow. In the time it takes to fetch one piece of information, a modern processor could have executed many hundreds of instructions. Clearly this makes the memory system a significant performance bottleneck. While there have been a large number of methods used to cope with this bottleneck, the most common and pervasive is caching. As shown in Figure 2.1, a small, fast memory subsystem, the cache, is placed between the processor and the main memory. The purpose of caching is to keep the most commonly accessed information in a closer (and much faster) location, the cache.

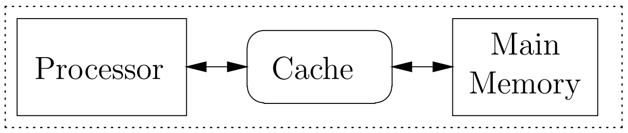


Figure 2.1: A simple caching structure.

The cache generally keeps a copy of the most recently referenced information found in the main memory system. While this cache is usually orders of magnitude smaller than main memory, it is common to find that well over 90% of all memory references are in the cache [39]. The time it takes to fetch a piece of information from the cache is usually similar to the time it takes to execute a single instruction [17]. Obviously, this greatly speeds up the average time required to perform a memory access, and reduces the "memory bottleneck" mentioned above.

It is instructive to include more detail at this point. Each byte[2] of memory generally

[2]A byte generally consists of 8 bits and can be thought of as a single number in the range of 0 to 255.

has a certain numeric *address*. Normally the processor will request one to eight bytes of information at a time. If the information is not in the cache (this event is called a *cache miss*), the cache requests it from main memory. Once retrieved from main memory, the cache provides the information to the processor and keeps a copy. If the cache does have the information (a *cache hit*), it simply supplies its copy of the information.

In the remainder of section 2.1 goes into greater details about caching, but not beyond what can be found in any standard computer architecture book. Books by Hayes [38] and by Hennessy and Patterson [39, 56] in fact contain this information in significantly more detail. Interested readers may find that Smith's seminal work [64] on caching provides a complete, if dated, overview on caching. Also, books by Przybylski [57] and Handy [37] are outstanding. Przybylski's book, while published in 1990, provides an architectural viewpoint similar to that of this author. Handy's book is written from a circuit designer's viewpoint.

2.1.1 The structure and organization of a standard cache

In general a cache consists of two parts, the *data* part, which is the information stored in the cache, and the *tag* part, which indicates which address the data came from. Each tag consists of enough information to identify from which address the data came. When a memory access occurs one or more tags are searched and compared to the memory address of the memory access. If a tag matches the address of the memory access, the data associated with that tag is provided to the processor by the cache rather than requiring an access to the (much slower) main memory to get the same data.

The one-to-one relationship between tags and data is illustrated in Figure 2.2.

Ideally the cache could keep the data is the most likely to be reused. In practice, however, this is usually not viable. Imagine what would happen when the processor requests data

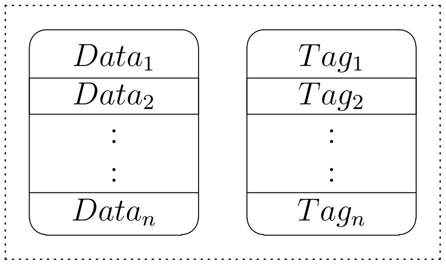


Figure 2.2: A cache with data and tags.

from the cache. The cache would need to search though each of the tags to find where the data is in the cache (if it is there at all). Since the whole point of the cache is to speedup access time, the time required to search all of the tags is usually not acceptable. A common solution to this problem is to assign each memory location a single place in the cache where it can reside. Since the main memory is much larger than the cache, many memory locations share the same assigned place in the cache. This means that only one of those memory locations could be in the cache at any given time. The net result is that flexibility is traded away for speed. This type of cache is called a *direct-mapped* cache. A cache where the data may be kept anywhere is called a *fully-associative* cache.

A compromise between the direct-mapped cache and the fully-associative cache is the *set-associative* cache. A set-associative cache is broken into sets of locations, all of equal size. Rather than mapping each main memory location to a specific line in the cache, the memory location is mapped to a set of lines. A given main memory location is now restricted to a very small number of locations in the cache. This increases the flexibility of placing data into the cache while still greatly reducing the number of locations which need to be searched. Caches where the set size is of size n are referred to as "n-way associative caches." Figure 2.3 illustrates the organization of these three caches. One can think of both a direct-mapped cache and a fully-associative cache as extremes of a set-associative cache.

In the remainder of section 2.1 we use the terminology associated with set-associative caches when referring to any of these three standard cache types.

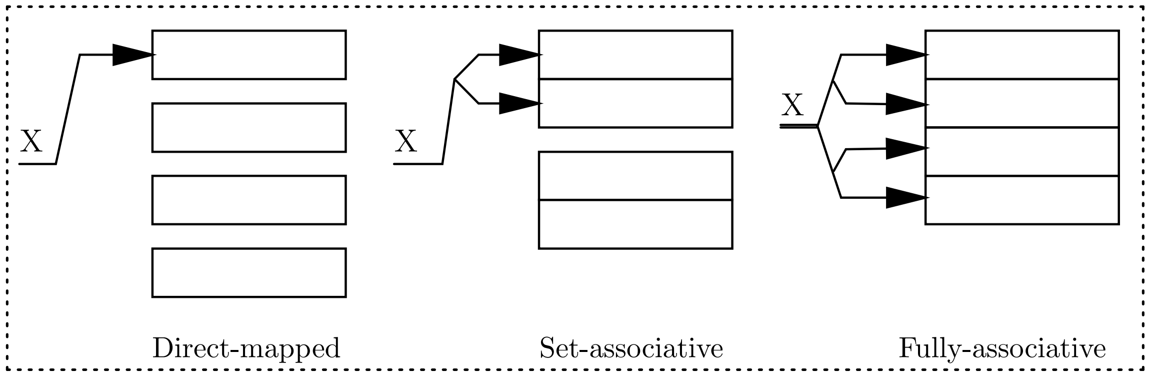


Figure 2.3: The three types of standard caches. If X is some address, this figure shows the different locations its cache block could be placed.

2.1.2 Address mapping

A standard cache of C bytes consists of K *cache lines*, each B bytes long. Further, there are S sets of cache lines, where each set consists of N cache lines. It follows that $C = K * B$ and $K = S * N$. In this section it is formally described how the various memory addresses are *mapped* into a cache. The next few paragraphs are summarized by Figure 2.4, and referring to that figure may be helpful.

Because of a property called *spatial locality*, (discussed in detail in section 2.1.6) it is likely that one of the next memory references will be to a location near the current reference. Thus, it makes sense to fetch the data near the current reference into the cache. To accomplish this task, the entire memory space is broken into non-overlapping blocks[3] of size B, and the entire block where the current memory reference resides is brought into

[3]In this document the term cache line (or line) is used to describe the location in the cache while the data that goes into the cache line is called a cache block (or block). Some others use the term "block frame" for line while other, mainly older works, make no attempt to distinguish between the two concepts.

the cache. If all the address bits other than bits $32 - lg(B)$ to bit 31 are the same, the two memory locations are in the same cache block.

Mapping each cache block into a set of the cache is the next task. Because of spatial locality it is best to insure that two cache blocks that have addresses near each other do *not* get mapped into the same set. To insure this, the blocks are mapped into the cache in a round-robin like scheme, so that a given set has every Sth block mapped to it. If two memory locations have the same values in bits $32 - (lg(B) + lg(S))$ to $31 - lg(B)$, the two addresses are mapped into the same set. These bits are often called the *index bits* as they are used as an index to select a set in the cache.

Finally the remainder of the address, that is bits 0 to $31 - (lg(S) + lg(B))$, make up the tag. Because of the mapping into the set and block, the remainder of the address is trivial to determine, and thus the tag is a unique identifier of the memory address from which the block came. Figure 2.4 summarizes the way a cache uses the address of a memory reference to generate the set number and tag. The offset simply states where the data is inside of the cache line.

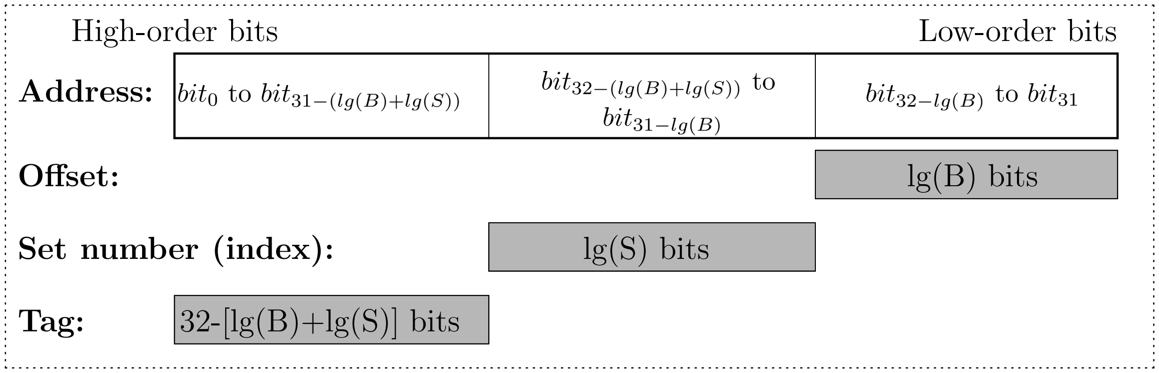

High-order bits
Low-order bits
Address:
bit_0 to $bit_{31-(lg(B)+lg(S))}$
$bit_{32-(lg(B)+lg(S))}$ to $bit_{31-lg(B)}$
$bit_{32-lg(B)}$ to bit_{31}
Offset:
lg(B) bits
Set number (index):
lg(S) bits
Tag:
32-[lg(B)+lg(S)] bits

Figure 2.4: How an address is used by the cache. B is block size and S is set size. Both are assumed to be greater than 1.

An example is likely useful at this point. Consider a 2-way associative cache which has 512 cache lines, each 16 bytes in size. Thus the data portion of the cache is 8192 bytes (or 8 Kilobytes) in size and there are 256 sets. Say the one byte of data at address $F0123456_{16}$ is requested by the processor. Notice that the offset of this address is 6_{16}, the set number is 45_{16} and the tag is $F0123_{16}$. Thus, the two tags associated with set number 45_{16} (69_{10}) are checked to see if they match the tag of the address $(F01234_{16})$. If either does, the data is found in the sixth byte (the offset) of the data part of the that cache line.

2.1.3 Replacement policies

If the cache is full and a new block needs to be put into the cache, some other block must be evicted. Clearly the block that is to be evicted must be from the same set as the new block. In a direct-mapped cache, there is no decision which needs to be made—there is only one block to chose from. However, in a set-associative or fully-associative cache, some decision must be made. By far the most common scheme is to evict the block which has been unused the longest. This block is called the least-recently-used (LRU) block and the algorithm is called the *LRU* replacement algorithm.

The reason the LRU algorithm is so commonly used is that programs tend to access data they have accessed recently. This property of programs is called *temporal locality* and is discussed further in 2.1.6.

2.1.4 Modern implementations

Modern caches are actually more complex than have been described above. In this section we outline some of the techniques used in modern memory system design.

Modern processors generally have multiple levels of cache. If the first and smallest cache (called the L1 cache) does not have the requested block, it checks in the next level of the cache (called the L2 cache). In theory this would continue until the last level of cache has missed, when the data would be retrieved from the main memory. Today, most general-purpose processors have only two levels of cache, although processors with three levels, and even four levels of cache are starting to show up in the marketplace[35].

Another quirk of modern caching is that the L1 cache is often split into two parts, one for data and one for instructions. This is done to improve bandwidth, improve miss rates, and for several other reasons[39]. Usually the data and instruction request stream are combined at the L2 cache. The caching scheme where the data and instructions are split is often called the *Harvard Architecture*[39], and is illustrated in Figure 2.5.

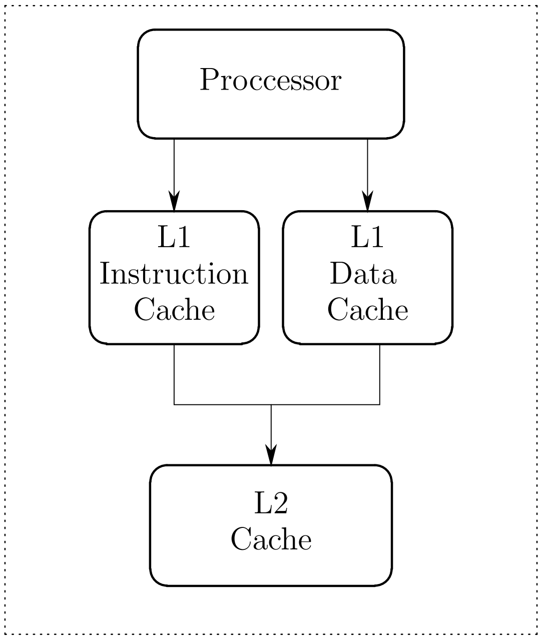


Figure 2.5: The Harvard architecture.

Lastly the use of virtual addresses has an impact on caching, particularly non-standard caching schemes. The details of virtual addressing can be found in most computer architecture and operating system texts[39, 63]. The net effect is that the address the processor uses (the virtual address) is different than the address the main memory system uses (the physical address). In general, the n least significant bits will not change when the virtual address is translated into the physical address. The value of 2^n is then called the *page size* and is often around 2^{12} bytes in size.

The point at which the virtual address is translated into a physical address has a significant impact on the cache. This translation can happen before, after or during the cache lookup. While this decision has a significant impact on cache performance, the only option which impacts this discussion is if the translation happens during the cache lookup. In that case, the index bits of the virtual address are used to select a set. The virtual address is translated as the cache is searched and the tag(s) are compared against the resulting physical address. This scheme has a number of names, including "virtually addressed, physically tagged," "virtually indexed, physically tagged" or simply a V/R cache [79]. Notice that it relies upon the index bits being the same for both the physical and virtual addresses[4]. This observation implies that the number of sets multiplied by the line size cannot be larger than the page size. The quick translation of virtual to physical addressing is done by a device called a translation lookaside buffer or TLB.

2.1.5 Caching metrics

The previous sections have hinted at the two basic metrics for measuring the goodness of a cache: *hit rate* and *access time*. Hit rate (HR) is simply the percent of accesses which are cache hits. The access time is the amount of time it takes to service a hit (T_{hit}) or a miss (T_{miss}). From these metrics the *average access time* of a memory access can be computed. It is simply $HR * T_{hit} + (1 - HR) * T_{miss}$. Recall that the advantage of direct-

[4]Although Wu, among others, proposes ways around this restriction [79].

mapped caches is a faster access time (a lower T_{hit}) while they tend to have a lower hit rate (HR). Fully-associative caches usually have longer access times but a higher hit rate. A detailed, if somewhat out of date, analysis of these trade-offs can be found in section 5.3.3 of Przybylski's book[57].

T_{miss} is the time it takes for an access to be fulfilled if the cache does not have a copy of the data, so it has very little to do with the cache itself. Rather it is dependent upon the memory system behind the cache. In fact T_{miss} can be an important parameter when designing a cache. If T_{miss} is high, increasing the hit rate at the cost of T_{hit} will be more beneficial than if T_{miss} were lower.

2.1.6 Locality of reference

Locality of reference is the fundamental reason a small cache can achieve a high hit rate. A processor tends to access memory locations it has referenced recently (temporal locality) as well as locations near those it has referenced recently (spatial locality). A cache takes advantage of spatial locality by fetching a larger chunk of data then the processor requests (a cache block). As discussed in section 2.1.3, the temporal locality of a program is utilized by the replacement algorithm, and by the very concept of a cache.

The above definition of locality is fairly terse and certainly qualitative rather than quantitative. Unfortunately that definition is literally as detailed as any standard computer architecture text gets. In Chapter 3 a number of locality metrics from the literature are discussed. In Chapter 4 a method for quantifying and visualizing locality is introduced in great detail.

2.2 Non-standard caches

While section 2.1 details how the most common caching schemes work, there are a large number of variations. Those variations include changes in replacement policy, cache structure, and address mapping. These variations usually can be evaluated in terms of the two cache metrics mentioned above: hit rate and access time. Each scheme attempts to improve one of the metrics, usually sacrificing the other to some small extent.

Evaluating these non-standard caching schemes can be difficult. First, it is difficult to prove that a given scheme will really improve the hit rate—even increasing associativity can hurt in certain cases [72]. Thus anyone proposing a new scheme has to show that their scheme works well on most real programs, quite a tall order. In general the authors of these schemes use the SPECCPU benchmark suite[5] as a standard method of evaluation. Even using this rather large suite of programs as a standard for measuring cache performance does not always generate reliable results. For one thing, this benchmark suite is biased toward certain types of programs [23]. Another problem is that when evaluating a novel caching scheme, one generally needs to write the cache simulator from scratch. This can be error-prone and has resulted papers full of errors. For example in an e-mail exchange with D. Rhoades, he agreed with this author that at least parts of their published work [59] suffers from significant simulator error.

The other difficulty is evaluating the access times of a novel caching scheme. The problem is that evaluation is an engineering decision where the quality of design, the processor layout, and the properties of the process technology used will have significant impacts on the actual access time. There have been high-level models proposed to estimate the impact of certain changes at this level [25]. However, even if those models are accurate, many

[5]Information about SPECCPU can be found at http://www.spec.org.

non-standard caching schemes do not easily fit into the parameters of these models.

Keeping these problems in evaluation firmly in mind, it is worthwhile to point out that just because these schemes are difficult to evaluate does not mean that they are not useful. Rather it means that when considering them one needs to keep a degree of healthy skepticism about the reported results. Independent measures of the hit rates of these caches have been reported [14, 60] and are probably somewhat more reliable results.

2.2.1 Hash caches

Perhaps the simplest and most common modification to a standard cache is to modify the way addresses are mapped into the sets of the cache. Rather than using the $lg(S)$ bits, where S is the number of sets, to select a set of the cache (as described in section 2.1.2), a larger group of address bits are used as inputs to a function which selects the set. This function is called the *hashing function*, and the cache which uses it is called a *hash cache*. For example if there are 2^m sets, one might use $2m$ different address bits and then perform a bit-wise XOR of those values to get an output of m bits. Figure 2.6 illustrates this hashing function. A more generic hashing cache was conceptually proposed by Smith in a 1978 work [65] while at least one real processor, Hewlett-Packard's PA7200, uses a hashed L1 cache [17].

The benefit of the hash cache is that it can reduce set conflicts in certain situations. Imagine that a program is *striding* [39] through memory 1024 bytes at a time. That is, it might reference location 0, then 1024, then 2048, etc.[6] Now imagine that we are using a direct mapped cache with 32-byte cache lines and a total of 4096 bytes in size. That cache has $4096/32 = 128$ cache lines. As the program strides through memory it will only utilize

[6]This type of behavior is quite common and usually associated with arrays.

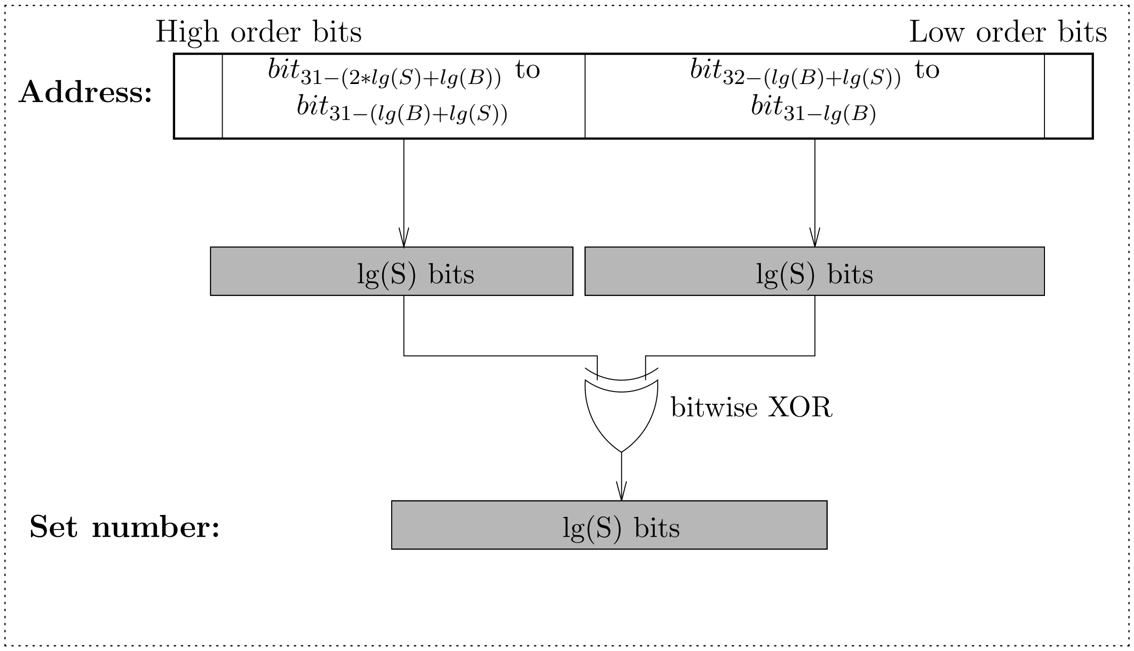

High order bits
Low order bits
Address:
$bit_{31-(2*lg(S)+lg(B))}$ to $bit_{31-(lg(B)+lg(S))}$
$bit_{32-(lg(B)+lg(S))}$ to $bit_{31-lg(B)}$
lg(S) bits
lg(S) bits
bitwise XOR
Set number:
lg(S) bits

Figure 2.6: An example of how the set number might be determined in a hash cache.

four of those cache lines, specifically lines 0, 32, 64, and 96. The hashing function described above will allow the cache to utilize all 128 cache lines.

A hash cache also has a number of restrictions and drawbacks associated with it. Performing the hashing function can take a significant amount of time. While a single XOR gate delay may seem trivial, that extra time may not be available[7]. Also, as described in section 2.1.4, if the cache is virtually addressed but physically tagged, the extra bits used by the cache to perform the hashing function may not be available until after the translation from the virtual to the physical address[60].

[7]In modern processes wire delays are starting to dominate the processor latencies. This means that the delay introduced by an XOR may become less significant in the future. [4]

One's complement cache

A one's complement cache [81] is a hash cache with a specific hashing function. Specifically, in the case of a cache of size 2^m, the address (other than the block number bits) is broken into groups of m bits. These fields are then XORed together. That creates a mathematical function similar to what would occur if the cache were of size $2^m - 1$ rather than 2^m, which can greatly reduce striding problems. The authors of the paper which introduced the one's complement cache claim that miss rates tend to drop 10% to 20%. Other studies indicate that the improvement is much lower for most traces, usually on the order of 1% to 3% [14]. However, we have found that in certain data-sets, a one's complement cache can have a significant impact, sometimes reducing miss rates by a factor of 10. While this may be due to a horrible compiler, it is certain that horrible compilers exist in the real world.

2.2.2 Hash-rehash caches

While the hash cache tries to reduce the miss rate of the cache, a hash-rehash cache[1] tries to decrease the time it takes for the cache to supply the data in the case of a hit (T_{hit}). This reduction is accomplished by addressing the cache as a direct mapped cache. If this access results in a miss, one of the index bits[8] is toggled, and the resulting value is used as the index to search the cache again. Notice that the toggling of a single bit means that two cache lines are always paired although the search order will vary. In this section we refer to X_1 as the first line searched for a given memory reference and X_2 as the second line searched.

If a hit occurs when searching X_1 the data is supplied to the processor and the cache is unchanged. If a hit occurs when searching X_2, the cache block there is swapped with

[8]Usually the most significant index bit. In any case, a given hash-rehash cache will always use the same toggle bit.

the one in X_1. If a miss occurs on both accesses, the block in X_1 is moved to X_2, the one that was in X_2 is evicted, and the new block is moved into X_1 The idea is to keep the most-recently-used (MRU) block in the cache line where it will be searched for first in an attempt to reduce the average value of T_{hit}.

The advantage of the hash-rehash cache is that each access takes as long as a standard direct-mapped cache, so if the first search is successful, the access time is reduced compared to a two-way associative cache. At the same time, the hit rate, when considering both tries to find the data, is better than a direct-mapped cache (but worse than a two-way associative cache[1]).

It is worth noting that the name "hash-rehash" cache is perhaps inappropriate. The index into the cache is usually done by bit selection, that is by using the index bits as described in section 2.1.3, rather than using a more complex hashing function. The reason for this indexing scheme is that even a short XOR delay might negate the access time advantages of a direct-mapped cache over a two-way associative cache.

While papers on hash-rehash caches and their variants [1, 3, 16] have been common, we are unaware of one being used in a commercial product[9]. It is difficult to say exactly why this is, but there are some obvious possibilities. Clearly the lack of index bits in a virtually addressed, physically tagged cache is an issue, in part because an X-way set-associative cache requires $lg(X)$ fewer index bits than a hash-rehash cache of the same size. Further, the access time advantage of a hash-rehash cache is going to be very dependent upon the exact technology used as well as other clock-limiting paths in the chip. One might be trading a lower hit rate for no actual improvement in access time.

[9]Although the IBM CMOS System/360's cache organization is somewhat similar to a hash-rehash cache and is described below.

Column-associative caches

One variant on hash-rehash caches is the column-associative cache[3]. A “rehash” bit is added to each cache line, which both improves the overall hit rate of the cache and reduces the average number of times the cache needs to be probed. As above, two paired cache lines are given labels, but this time the labels A and B are used and no order of access is implied. The rehash bit associated with a given cache line is set if it is accessed as a rehash location, either during a search or during replacement. If a cache line with its rehash bit set is searched, no attempt is made to search the rehash cache line. Say that lines A and B both hold ‘hash blocks’, that is blocks that would be found during the first pass of a search. In that case neither rehash bit will be set, so if a search of A fails, B will be searched. During the search of B (which will miss) its rehash bit is set. During replacement the block in A will move to B and the block in B will be evicted to make room for the new block. Now if B gets searched first, it will result in a miss (it holds data associated with A) but A will not be searched (because the rehash bit in B was set). The new data, when fetched, will be placed in B and its rehash bit will be reset.

The net effect is two-fold. First, a more LRU-like replacement scheme is implemented, resulting in a better hit rate than a hash-rehash cache [3, 82]. Second, in some cases a miss does not require two searches through the cache, slightly reducing both the average access time and the number of requests to the cache.

Other hash-rehash schemes

There have been a number of other proposed schemes related to the hash-rehash cache. Detailing each is outside the scope of this document, but short descriptions of a few are provided in this section.

The predictive sequential associative cache [16] is very similar to the column associative cache. However, instead of swapping blocks to keep the MRU block in the first location, a steering bit table determines which location to search first. It also maintains true LRU ordering with a single LRU bit. A number of schemes for predicting the steering bit were also proposed.

There are also two caches called the MRU cache, both of which have similarities to hash-rehash caches. The first was used by IBM's CMOS System/370 [18]. Here they kept the MRU data of each set on chip, and had the rest of the cache off-chip. They then speculatively used the MRU data whenever the cache was accessed. Because the time to detect the mistake was always the same, the logic required to back out of a misprediction was fairly straight-forward. The other MRU cache [47] simply did a sequential search through a set of the cache, using MRU information to guide the search order.

2.2.3 Victim caches

The victim cache is perhaps the most exciting non-standard cache thus far proposed. As originally proposed [45] it has no impact on the access time of an L1 hit (T_{hit}) or on the hit rate (HR). Rather it reduces the average access time for an L1 miss (T_{miss}). The only price is a small bit of additional die area.

A victim cache is simply a very small (usually 2 to 16 cache lines) fully-associative cache. It acts just like an L2 cache managed with an exclusive policy [9], only rather than the normal latency associated with an L2 cache, the small size allows the access to occur in about 1 cycle. So when the cache evicts a block, rather than moving that block to the L2 cache (or just throwing it away) it is moved into the victim cache, and a block from the victim cache is evicted to the L2. Thus, the main L1 cache and the victim cache never store the same block. Even very small victim caches can expect to recover 30% or more of the conflict misses that occur in a direct-mapped L1 [45]. The write buffers in Intel's Pentium Pro as well as the "victim buffers" of the Alpha 21264 both perform some of the functions of a victim cache.

It is worth noting that due to the small size of the victim cache it is quite reasonable to search the victim cache in parallel with the main L1 cache. While this will slightly increase the total latency of the cache, we believe that the access time will be similar to a two-way associative cache, assuming that the victim cache size is kept quite small.

2.2.4 Skew caches

Only a small modification to a set-associative cache is required to create a skew cache. As illustrated in Figure 2.7 each bank in a set-associative cache can be thought of as a direct-mapped cache with each of the banks indexed by the same function. In a skew cache a different function is used for each bank. A 2-way skew-associative cache is illustrated in Figure 2.8. In this case two different functions, $f0$ and $f1$, are used and a block may be placed in different locations in different banks.

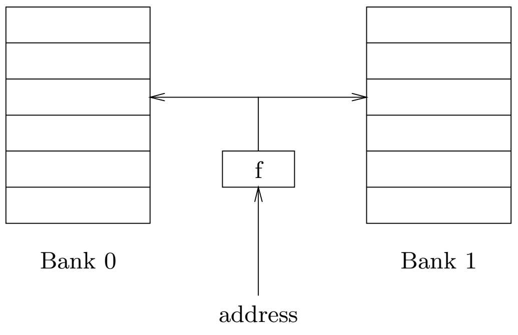


Figure 2.7: A model of a 2-way set-associative cache. [60]

Such a simple modification may not seem significant, but studies have shown that a two-

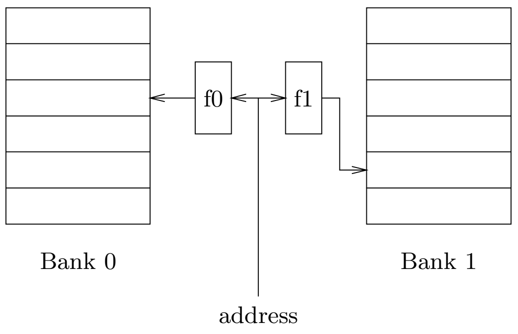


Figure 2.8: A 2-way skewed-associative cache. [60]

way skew-associative cache produces hit rates similar to that of a four-way set associative cache [14, 60, 61]. This performance can be attributed to three effects: data dispersion, the effect of hashing, and self-data-reorganization [14, 61]. The first effect creates an actual increase in associativity [14]. That is, temporal locality is more helpful to a skew cache than a standard cache of the same associativity. The second effect is an identical effect to that found in the above section on hash caches. The third effect is more of a side-effect of locality. A cache block can go into two different lines. If one line is an area of high-conflict, the block will likely get evicted. Next time it comes in it may go to the other line and, due to the lower conflict, stay there. Thus, on relatively slow changing data sets, skew caches can "learn" where to place blocks.

2.2.5 Smart caches

All of the caches thus far discussed, both standard and non-standard, achieve their high hit rates purely by taking advantage of locality. While high hit rates are achieved by these caches, there are limits to how successful a cache can be by purely relying upon locality. As these limits are discussed in section 2.1.6 of this document, they will not be dwelled upon here. However, the basic observation is that even a fully-associative cache has non-compulsory misses, and no LRU cache can be expected to perform better than that.

Smart caches[10] are those caches whose caching schemes rely upon something other than just locality of reference. Many of these schemes rely upon the observed fact that certain accesses are unlikely to be re-referenced in the near future [51, 78], and having those references *bypass* the cache may be the best solution. By bypassing these low locality references other, ideally more-likely-to-be-used, cache blocks can be kept in the cache.

Each of the smart cache techniques utilizes different ways of identifying the low-locality accesses. Additionally, many of these techniques provide methods for reducing the impact of bypassing accesses which have a high degree of locality. We are unaware of any studies which compare these smart caching schemes. The techniques used by these schemes are so diverse that such a study would be extremely difficult to conduct, and even more difficult to conduct fairly. Many of these techniques have not made it past the "proof of concept" stage—comparing them to more refined methods would not show that one idea was inherently better than the other. While few quantitative comparisons seem to exist, it is possible to detail how some of these schemes work.

Assist caches

Hewlett-Packard's PA7200 utilizes a smart caching scheme which can be referred to as an assist cache [17]. Those familiar with HP's recent endeavors will not be surprised that the identification of the low locality memory references is left to the compiler. In fact the assist cache, for which this scheme is named, is little more than the backup in case a high locality access was misidentified.

[10]The term "active management" is perhaps a more common term than "smart caching." However, that term seems to imply a dynamic hardware caching scheme, and we wish to also include compiler based management.

The PA7200 has two L1 data caches. The first, called the assist cache, is a physically-indexed, 64-entry, fully-associative cache using a FIFO replacement scheme. The second is a virtually-indexed, physically-tagged, hashed, direct-mapped cache and can be as large as one megabyte. In general, data is first put in the assist cache and evictions from the assist cache are placed in the main cache.

The compiler can identify a certain load or store as being, in HP's terms, "spatially local." What they mean is that the compiler believes that the line may be accessed again due to spatial locality, but not due to temporal locality. Lines so identified are still put into the assist cache, but upon eviction the main cache is bypassed. This backup by the assist cache allows the compiler to be aggressive in its attempts to identify low locality accesses.

Stream buffers

Another method of identifying low-locality accesses is the hardware stream buffer [45, 51]. A stream buffer is basically pipe, that is a FIFO queue, where information can only be taken off the top of the pipe. By some method, either with information from the compiler or observation of the data stream or both, streams of references are discovered. Streams of references are generally individual load instructions walking though memory with a known stride. The data can be prefeteched into the FIFOs and the caches can be bypassed entirely.

There are many advantages to stream buffers. For one, traditional prefetches into a cache can result in data being evicted before it is used due to conflict in a certain set. Secondly, streaming data usually has very limited temporal locality, and steam buffers reduce the pollution of the cache by this low-locality data [51]. Also, modern DRAM components haven't been truly random access since the introduction of fpDRAM [42]. For modern DRAMs, spatial locality can be very important indeed. Reordering the memory accesses so that large chunks of each stream are fetched at once, rather than the program order of interleaved access, can greatly reduce the total latency to fetch data from the DRAM [52].

There are some potential problems with stream buffers. While stream buffers do show significant performance increases on certain programs, non-scientific programs tend to see more limited benefits. This characteristic makes their use questionable in general-purpose processors. Also, stream buffers, unlike most of the caching schemes discussed so far, generally require compiler help to achieve any significant performance gain. That means that effort by the chip designers is not enough; effort is required by the compiler writers too. It is worth noting that while we are unaware of a commercial chip that uses stream buffers, an experimental chip with stream buffers has been built at the University of Virginia [50].

Cacheable/Non-Allocatable Cache

It can be argued that the true beginning of smart caching algorithms was a paper by Tyson *et al.* [74]. In that paper it is shown that not only do certain load instructions fetch the vast majority of the data, but some load instructions are responsible for a disproportionate amount of the cache misses. This observation means that identifying those instructions can provide large hints about which memory references should bypass the cache.

They propose a number of techniques for identifying load instructions which should be bypassed, including compiler/profiling methods and dynamic two-bit schemes that resemble branch predictors. While their proposed schemes do not improve the average hit rate, they do show that memory system bandwidth can be reduced. More importantly this paper provided the fundamental basis for investigation of *instruction*-based load bypassing. This scheme is called C/NA or CNA as the bypassed loads are marked as “cacheable but non-allocatable,” meaning that a given cache block may be in the cache but, if it is not, do not allocate space for it in the cache.

Other smart caching schemes

A number of other smart-caching schemes have also been proposed. Some of the more interesting are briefly described in this section.

A rather novel hardware scheme for smart caching was proposed by Johnson and Hwu [44]. The underlying idea is that certain data structures are, by their very nature, accessed in a low-locality manner. They propose to group the entire memory space into macroblocks and observe the reuse behavior of the various macroblocks. If a given macroblock has low reuse, the data from that block bypasses the cache. Under one version of their proposal they add a small buffer to act as a back-up in the event of a misprediction. The initial results of their study were encouraging, but covered a very limited set of programs[11]. In the literature this scheme is often called the MAT cache [69].

A research group at the University of Michigan has proposed a number of smart caching schemes. Their NTS cache is is similar to the MAT model in that the data addresses are used to predict the locality of a memory location. However, in the NTS cache the metric of locality considers whether the cache block had been used more than once in its previous "tour" of the L1 cache, rather than looking at a macroblock over a much longer time [70]. That some group has also improved the CNA scheme.

Wong and Baer proposed a scheme to improve the hit rate of a highly associative L2 cache [78]. They tag each load instruction as either generating "temporal" cache lines or "nontemporal" cache lines. A standard L1 replacement policy is used, but the L2 gives preference to temporal cache lines. They propose both a compiler-based profiling scheme

[11]For example, programs like TPC-B have a *very* different access pattern than this scheme assumes.

and a hardware scheme to distinguish the two types of load instructions. Their results look both complete and promising, showing as much as a 12% improvement in instructions per clock for some programs, although 1 to 3% seems to be the norm.

Anderson *et al.* study the impact of *bypassing* and *superloading* on a miss. A *superload* moves both the missed line and the surrounding lines into the cache [5]. They propose a hardware scheme that decides to load or superload based upon past behavior of that part of memory. They compare their on-line scheme to the off-line optimal algorithm, both with and without bypassing. The net result is that their scheme shows marked reduction in both the miss rate and bandwidth used when compared to a more traditional LRU cache. However, their scheme also utilizes an unlimited amount of memory to maintain information about previous loads. The net effect is that they show superloading and bypassing to have significant potential in an on-line environment. However, their exact implementation appears infeasible at this time.

Hallnor and Reinhardt have recently proposed a much more complex smart caching scheme [36]. The fundamental argument they make is that memory is getting *so* slow compared to the processor speed that it is wise to take extra time on an L2 access in exchange for a higher L2 hit rate. While this premise may be flawed[12], their results are quite promising. Just as others have done, they apply a replacement scheme that uses past history. However, they also use techniques from operating systems to make the L2 cache effectively fully-associative. This associativity comes at higher system latency, but allows them to use their replacement scheme more effectively. The number of misses is greatly reduced, sometimes by as much as a factor of two.

[12]For example, L2 misses, if rare enough, might be dealt with in other ways, including changing threads or processes

2.2.6 Further reading

While citations for each of the above non-standard caches have been provided, some of the papers provide outstanding explanations of a wide variety of schemes. In particular [70, 82] provide very nice overviews of hash-rehash caching and smart caching respectively.

Chapter 3

On the Measurement of Locality

The purpose of this chapter is to provide an overview of research in the area of locality measures. In the first section of this chapter, the meaning of the term “locality measure” is discussed. In sections 3.2 and 3.3 various locality measures are classified and explained. Section 3.4 contains a discussion about *LRU stack distance*, both as a locality measure as well as some of its other applications in the literature.

3.1 Locality measures

Measuring the locality of a data stream is an imprecise science. In general, it is an attempt to reduce massive amounts of data, sometimes millions or billions of data points, into something which can be easily understood and used by a human. Each researcher has different goals and hopes for their measure, so each researcher proposes a different scheme. In fact, the motivation for each measure is perhaps as interesting as the measures themselves.

3.1.1 What is locality?

The exact definition of locality is a bit vague, which, as one would expect, contributes to the difficulty of measuring it. However, the term locality, as was briefly mentioned in Chapter 2, is used describe the fact that there is a dependence between the addresses of previous memory accesses and that of future accesses. A processor tends to access memory locations it has referenced recently (temporal locality) as well as locations near those it has referenced recently (spatial locality). This phenomenon is to be expected—as a program moves through a data structure it will do so in a certain pattern; perhaps "walking" memory at a fixed stride size, or perhaps moving though a linked list or tree. In reality, a program will generally interleave its accesses to various data structures.

As an example, imagine a standard matrix multiply operation. One input array is accessed in row order, one in column order, and the output array is usually accessed in either row or column order. In addition, loop control variables and temporary variables are heavily utilized (although many of those may be allocated to registers and not manifest themselves as memory accesses). Now imagine that the matrix multiply operation is just one of a number of subroutines used by the program. The net effect is that a number of access patterns start, get interleaved with other access patterns, and then end. These access patterns are much like threads in a tapestry. When examining that tapestry what should be the goal? One could evaluate the individual threads, the patterns formed by those threads, or perhaps the tapestry as a whole. Each level of granularity is important, and each requires a different perspective.

3.1.2 Locality measures in the literature

The remainder of this chapter summarizes much of the existing literature about locality measures. Section 3.2 provides an overview of the most common way of examining locality: at the macroscopic level. These metrics are generally concerned with capturing the locality of an entire program, and generally rely upon the law of large numbers to function correctly. In that section a brief overview is also provided for a closely related topic, cache-centric locality models. Those models view locality as it is reflected in a cache.

Section 3.3 looks at those locality metrics which are generally concerned with a much smaller number of references. These microscopic metrics tend to be most useful when examining certain program constructs or data structures. Many of them are loop-centric. That is they assume the existence of a single large looping structure with relatively regular behavior, something generally found in scientific computing.

Finally, in Section 3.4 the idea of LRU-stack distance is introduced. In addition to other applications, this scheme is clearly a macroscopic locality measure. It is was one of the first of the locality measures of any type, and has a large number of applications. In Chapter 4 this measure is discussed in great detail and novel applications for it are introduced.

3.2 Macroscopic locality measures

Macroscopic locality measures are generally concerned with characterizing the locality of a program or a large portion of a program. Most of the papers in which these models are discussed focus upon predicting the cache miss rate. Their goal is to allow the quick characterization of a large number of programs on a large number of caches in a short period of time. The huge increases in disk space and processing power, as well as the high speed of the cache simulators such as Cheetah [68], have made this less important, at least for standard caches, than it was in the past. However, by attempting to predict a miss rate these studies had to generate a model for locality. It is that part of their work, the locality model, on which we concentrate in this section.

3.2.1 Chow's empirical power law

Perhaps the simplest useful model of locality was proposed by Chow in the mid 1970s [19, 20]. He proposed that the miss rate of a cache could be matched to the equation $M(C) = C^{-\alpha}$, where α would be fit to the available data and where C is the capacity of cache. Chow claimed that the miss rate is "most sensitive to the capacity" and that in his analysis issues such as "block size, the management algorithms and other factors" would be ignored. If the cache is considered to be a fully-associative cache using an LRU replacement algorithm (something his analysis would certainly allow), $M(C)$ is precisely a prediction of temporal locality as it is defined in Chapter 5.

While outdated and lacking significant quantity of empirical data to support it, Chow's power law is very important to the study of locality. First, it forms the basis for one of the classic rules of thumb in cache design: increasing the cache size by a factor of 4 results in one half the miss rate. This would be the case if α were 0.5. This general rule, if not the 0.5 value, is experimentally observed in [64] among others.

Chow used his power law as the basis for a mathematical approach to the design of multiple levels of the memory hierarchy [20], a research topic that has continued well into the 1990's [43]. While it is his power law that is relevant to our work, the primary result of [19, 20] was to show that the optimal number of cache levels scaled logarithmically with the capacity of the cache hierarchy.

3.2.2 The Fractal Model of Locality

Thiebaut's work [71] on the fractal model of locality is something of a variation on Chow's [20] power-law model from the decade before. How he arrives at his model is very different than Chow. Thiebaut claims that the access patterns of many programs are fractal in nature: having an access pattern in one area of memory and then jumping to a different area of memory and repeating that same pattern. He claims that if this is true, a two piece power-function, very similar to Chow's, should model the ratio of the unique number of memory locations over the total number of memory references. His function, $f(x)$, takes as an argument a number of successive memory references x and predicts how many different locations those x memory references will access.

The fundamental argument is that memory accesses are in fact a random walk though memory. Also, the "jump size" of that random walk tends to be low, but has the occasional long jump in it. Thiebaut states that such a random walk is a special case of a fractal, and that other work in fractals can be used to model this access pattern, and thus the locality of the trace. He admits that other than in the instruction stream, much of the random walk theory is not supported by his data. He believes that this is due to memory accesses actually consisting of many different random walks. We suspect that examining the data stream generated by each individual load and store instruction might provide a way to observe these random walks, if they do indeed exist.

What Thiebaut does demonstrate is that assuming the random-walks are occurring, he should be able to predict the miss rate of a fully-associative LRU cache. His model has two parts: one for the first n_0 memory references, and one for the remainder of the accesses. In effect he produces a formula of the form $MR(C) = AC^{\theta}$ with different values for A and θ for accesses before and after access number n_0.

The results can be criticized for only addressing a very small number of traces. We believe his model is of marginal value, but the underlying idea of data streams being modeled by a random walk is of interest, and may have applications to stream detection and prediction found in [51] and related works.

3.2.3 Agarwal's analytic cache model

In 1989 Agarwal *et al* presented a novel analytic cache model [2]. Although this work is presented as a model of cache behavior, it can be viewed primarily as modeling the locality of a reference stream. Their goal is to describe, with as few parameters as possible, the various properties of the reference stream. They attempt to quantify temporal locality, spatial locality, and the degree of conflict in the reference stream. The reference stream is broken into "time granules," and values are found for each parameter in each time interval as well as for the entire reference stream.

Ignoring multi-programming related issues, this analytic model has four locality parameters. The first is the unique number of blocks that are accessed. This provides a measure of the degree of temporal locality very similar to that used by Thiebaut. The second parameter is the *collision rate*: an attempt to measure how prone a given reference stream is to evicting live blocks[1]. The final two parameters are used as inputs into a two-stage Markov model. The Markov model is used to describe the probability of memory accesses being to successive locations. It is this measure that is used to account for spatial locality.

Their scheme does have its flaws. First, for computational reasons, it relies upon the cache using random replacement rather than LRU replacement. While they do claim that the miss rate for random replacement is a good indicator of a cache with LRU replacement,

[1]That is, blocks that will be reused.

neither their data, nor this author's experiences support this. They report that the model does a reasonable job estimating the miss rate for direct-mapped caches, with an average error of 4% to 15%, depending on the block size. However, two-way set-associative caches have an average error of 14% to 23%. They also note that their model always predicts too high of a miss rate for the set-associative cache and attribute that behavior to the random replacement policy assumed by their model. While their error rate might seem unacceptably high, it should be noted that the space needed to store this analytic model is at least three orders of magnitude smaller than the trace itself. Further, the results from this model can be computed *much* faster than actual simulation. Thus, this model is a reasonable alternative for fast design space explorations. The results from such an exploration could be used to determine which cache configurations and traces to study in greater detail.

3.2.4 Grimsrud's locality metric

In the early 1990's Grimsrud, in some cases with others, wrote a number of papers about quantifying locality [32–34]. Their work describes a method of visualizing locality with a three-dimensional graph. The x dimension is spatial locality, the y dimension temporal locality, and the z dimension indicates the probability of access having this locality value. Specifically, the locality of a memory reference trace $\vec{T}$ is defined to be $L(\vec{T}, s, d)$. This is the probability that an address s away from the currently referenced address is, in the remainder of the trace, first accessed d references in the future. More rigorously,

$$L(\vec{T}, s, d) = Pr(\vec{T}[x] + s = \vec{T}[x+d] \wedge (\vec{T}[x] \neq \vec{T}[x+1], \ldots, \vec{T}[x+d-1])$$

To make the graph readable, Grimsrud groups the values of s and d into bins. That is, each value of d and s is rounded up to the nearest power of two. This rounding not only gives a logarithmic view, it also makes the surface of the graph much cleaner and meaningful. This "binned" version of L is renamed L^* and is also called the *locality function*. By graphing the three-dimensional locality function for a given trace, one can gain an intuitive feel for the type and degree of locality that exists in the trace.

Applications

Grimsrud provides a wide variety of applications for his locality metric. One very useful application is in evaluating artificial trace-generation techniques[2]. While he states that the locality function cannot easily be used to generate an artificial trace, it is possible to check the correspondence between the locality functions in the artificially generated trace and the original trace it was based upon. Grimsrud evaluates a number of techniques, including a stack-based model [6, 8] and a Markov model, concluding that each of the models has certain flaws and demonstrates the weaknesses of each.

There are a number of other proposed applications [34] including trace characterization, benchmark evaluation, and the ability to visualize locality. The last is the most important in the context of locality measures, and Grimsrud explains how this feature can be used to help one identify loop characteristics including stride and the amount and nature of temporal locality in the reference stream.

3.2.5 Cache based locality

While Chow's, Thiebaut's, and Agarwal's models are cache models after a sense, in each case their results hinge upon their characterization of the reference stream in terms of a few parameters. The interaction of caches with their locality model can be viewed strictly as an

[2]Briefly, artificial trace-generation is an attempt to summarize the locality properties of a trace in a set of parameters. This is mainly used to avoid the need to store long traces on a computer [8, 33].

application of the locality model. However, there are also "pure" cache models that have some relevance to the study of locality. These models generally use "ideal" caches, which are unrealistic to actually implement, as a baseline for use in gauging the performance of a more realistic cache. In doing so they also provide a glimmer of insight into the locality of the reference stream being examined.

Perhaps the most widely used model of this type is the "three C's" model [23, 41], developed by Hill and popularized by Hennessey and Patterson in their two computer architecture texts [39, 56]. In this rather simplistic model, cache misses are broken down into three categories: *compulsory*, *capacity*, and $conflict$. Compulsory misses are misses to never-before referenced cache blocks. Capacity misses are those misses on which a fully-associative LRU cache of the same size would also have missed. Conflict misses are misses on which the fully-associative cache would not have missed. Clearly the compulsory misses tell us about the number of unique blocks in the reference stream and the capacity and conflict misses describe the degree of temporal locality available.

A model closely related to the three C's model was proposed by Sugumar and Abraham in 1993 [68]. They take issue with using a fully-associative LRU cache as an "ideal" cache, and consequently as the basis for comparison. Rather they suggest that more information could be gathered by using optimal placement algorithms. They instead divide cache misses into four categories: *compulsory*, *capacity*, *mapping* and *replacement*. Using their notation, let $M(L, C, k, repl)$ represent the number of cache misses, on a given trace, in a cache where L, C, k and *repl* are the line size, cache size, associativity, and replacement strategy respectively. Table 3.1 shows how their method categorizes those misses. C/L represents a fully associative cache.

Suppose we are evaluating a C-byte, k-way associative LRU cache. Compulsory misses

Miss type	Number of misses
Compulsory	M(L,any,any,any)
Capacity	M(L,C,C/L,OPT) - M(L,any,any,any)
Mapping	M(L,C,k,OPT) - M(L,C,C/L,OPT)
Replacement	M(L,C,k,LRU) - M(L,C,k,OPT)

Table 3.1: A classification of misses using the optimal replacement policy. The arguments to the function M() are line size, cache size, associativity and replacement policy in that order. L is the line size of the base cache, C is its cache size, and k is the degree of associativity.

would be those misses which occur because the cache block had never-before been referenced, a definition identical to that of the three C's model. Capacity misses, however, are defined to be those non-compulsory misses which would also occur in a C-byte optimal, fully-associative cache. Mapping misses are those misses which do not occur in a C-byte fully-associative optimal cache, but do occur in a C-byte k-way associative optimal cache[3]. A replacement miss is one which occurs in a k-way set associative LRU cache but not in a k-way associative optimal cache. Ideally then, capacity misses are those accesses on which a cache is truly too small to expect to achieve a hit. Mapping misses represent misses due to structural issues (conflict), while replacement misses are those which a better replacement policy would have avoided.

Sugumar and Abraham's scheme has many advantages over the three C's model. For one thing the three C's model assumes that an access which misses in the fully-associative cache also will miss in a less associative cache. As is shown by [58, 72] and others, this need not be the case. In fact, as we show in Chapter 4, an access which just barely qualifies as a capacity miss has over a 40% chance of actually being a hit in a direct-mapped cache. Because Abraham and Sugumar use OPT rather than LRU as their basis for comparison, this issue does not arise in their work. In fact the aggregate definitions in Table 3.1 can never

[3]Using our terminology from Chapter 2, we would call this a *structural miss* rather than a mapping miss

be negative, something not true of the equivalent three C's definitions [68]. Additionally, this OPT-based scheme provides a method of evaluating the replacement policy, something not addressed by the three C's model.

There are other cache studies and metrics which can provide information about the locality of the trace. *Liveness*, used by [15, 49, 54] among others, describes the percent of blocks in the cache which will be re-referenced sometime in the future (or perhaps within some fixed time window.) Additionally many of the smart caching studies have attempted to find the relationship between locality and the referencing instruction [74] or locality and the referenced location [44]. These were briefly described in Chapter 2.

3.3 Microscopic locality measures

As noted above, microscopic locality measures are concerned with relatively small numbers of memory references, often on the order of hundreds to tens of thousands. This section provides a brief overview of the various microscopic locality studies in the literature.

3.3.1 Instantaneous locality

Instantaneous locality, as proposed by Weikle *et al.*, is perhaps the most recent attempt in the literature to quantify locality [76]. They argue that caches *filter* locality, just as a lens can filter light or electronic filters can filter signal frequencies. Their goal is to quantify locality in such a way that this filtering can be observed and have a "high correlation to intuition."

An electronic signal is simply a changing voltage which has a single value at any given time. As such they propose that the locality of the reference stream should also have some value at any given time. The measure they tentatively propose for instantaneous locality,

l_i, of a reference a_i in a stream of references $(a_0, a_1, a_2, \dots)$ is:

$$l_i \approx \sum_{j=0}^{i-1} \left(\frac{1}{|a_i - a_j| + 1} \right) \left(\frac{1}{|j - i|} \right)$$

The first term, $1/(|a_i - a_j| + 1)$, corresponds to spatial locality. If the two addresses a_i and a_j are the same, this term will be unity, otherwise it will be (significantly) less than one. The second term, $1/|j - i|$ corresponds to temporal locality. If the two accesses are adjacent (that is $i = j \pm 1$), this term is unity, otherwise it is less than one. When these two terms are multiplied together a value ranging from nearly zero to one (for two adjacent accesses to the same location) is generated. The summation of these terms determines the value of the instantaneous locality of a_i.

Once a definition of instantaneous locality has been created, the next step is using it to see how well locality is filtered out of the reference stream. In order to visualize this, one other term, the *instantaneous hit rate* for reference i, is defined as $h_i = \delta_i + \sigma \dot{h}_{i-1}$ where $\sigma = 0.5$ and δ_i is one, if access i was a hit and zero otherwise. This gives h_i a range from zero to two, where higher values indicate a higher recent hit rate. With this definition firmly in hand, they show that accesses with high instantaneous locality values tend to have high hit rates and that the output locality stream (the requests made by the cache) generally has less locality than the input stream. This metric is also used to compare different cache types and to try to recognize program constructs from their instantaneous locality graphs [76]. Another application is that the "bursty" nature of real programs—real reference streams tend to have periods of high locality separated by periods of low locality [10]—can be easily seen and measured.

The authors of this work readily admit that the exact formulas they have selected may not be the best. Rather they defend the notion that instantaneous locality, as a concept, is both useful and important and better formulas will only improve the concept. This author would certainly claim that their formula for instantaneous locality greatly underestimates the importance of spatial locality. For example, two memory accesses that are only 8 bytes apart are given a spatial locality measure of 1/9 on a scale of 0 to 1, too low by far in a time when cache lines are often 64 bytes or larger. Another limitation of this work, and why we characterize it as a microscopic locality measure, is that viewing large reference streams is difficult. The largest example in their paper dealt with fewer than 4000 memory referees. Of course, with the aid of a computer and some additional techniques, visualizing much larger traces may be possible[4].

3.3.2 Loop-based locality metrics

A common form of locality analysis has been based on the study of loop-nests. While it is interesting, loop-based studies are only peripherally related to our own locality studies (found in Chapter 4) and as such will only be briefly discussed. Two papers have been chosen as representative of the field. McKinley and Teman's work [53] evaluates the applicability of common assertions about *program* locality in the context of *loop* locality, while Ghosh *et al.* [29], generate "cache miss-equations" which can be used to guide code optimizations in the compiler. In the remainder of this section some of the basic jargon of this field is introduced followed by a brief overview of the two selected papers.

In general, loop-based locality studies are targeted at compiler writers. Usually their purpose is to propose an algorithm for analyzing the memory reference patterns and find a way for the compiler to reduce the number of data cache misses. In doing so, these studies

[4]They have written a program which aids the analysis of instantaneous locality, but we have not had the opportunity to examine it in any detail.

have adopted interesting terminology. Memory references are described in two methods. The first method is the three C's model discussed in Section 3.2.5. The other method describes a non-compulsory reference with a three-word phrase consisting of hit or a miss; temporal or spatial; and self or group. So a given memory access might be a "spatial group miss" or a "temporal self hit" or any combination of the three terms. An access to address X by instruction I is called temporal, if the last time (which may be now) the block in which X resides was in memory was when X was referenced. Otherwise, the access is labeled as spatial. If the access was temporal and instruction I was the instruction which had last referenced X, the access is labeled as "self." If the access was spatial and the last instruction that referenced the block was I, the access is also labeled as "self." Otherwise, it is labeled as "group."

This jargon is useful for a couple of reasons. First, it is the only locality terminology of which we are aware that indicates something about the referencing instruction, an important concept for some smart caching schemes [74]. Second, it allows a high-level understanding of the degree of temporal and spatial locality, something which otherwise has required a more complex scheme such as [33]. It also considers the access patterns of hits as well as misses, something the three C's model does not.

Examples of loop-locality from the literature

McKinley and Terman [53] point out that most locality studies examine program locality, while compiler writers tend to concentrate on cache optimizations restricted to looping structures. They find that conflict misses are much more common in looping structures than capacity misses. Also, they observe that loops see more spatial reuse than the overall program. While neither of these results come as any surprise, they do provide further evidence that most of the cache misses occur between looping structures. This observation means that rather than focusing on the "low-hanging fruit" of intra-loop cache placement, inter-loop caching issues need to be examined more carefully by compiler writers.

Ghosh *et al.* [29] provide a much more traditional view of loop locality. They generate a set of mathematical equations for certain highly-restricted looping structures. These equations can be used to determine the optimal tiling and array padding required to minimize the miss rate. In creating these equations they have managed to quantify locality in a very different way than any other study considered in this thesis. While it is unclear if their methods would be useful to a human attempting to understand loop-locality, it is clearly useful for an optimizing compiler.

3.4 Stack distance

A program trace consists of the ordered set of addresses referenced by a program. Imagine pushing each memory reference onto a stack. If the reference had occurred earlier, it would be removed from its current position in the stack and then pushed onto the top of the stack. We define the stack distance of a reference R to be the depth in the stack from which it was fetched. If R is not on the stack (because it had never before been referenced) we define its stack distance to be ∞. This concept has been used in [8, 13, 33, 41, 49, 58, 64]. An equivalent definition of stack distance is to count the number of unique references which have occurred since the last reference to R.

The earliest paper which used stack distance in this context was by Mattson *et al.* [49], where they defined the top of the stack to be the 'one' location rather than the more standard convention of the top of the stack being the 'zero' location. Others followed suit, and for sake of consistency with previous work we adopt the same notation in the remainder of this chapter. Thus, the stack distance of a reference is the number of unique references which have occurred since the last reference to that location, plus one. Figure 3.1 illustrates the computation of the stack distance value.

Reference Stream:	A	B	C	C	B	A	D	C	A	A	B	D
Stack Distance:	3	2	1	4	4	3	4	∞	1	∞	∞	∞

Figure 3.1: Illustration of the stack distance of a reference stream

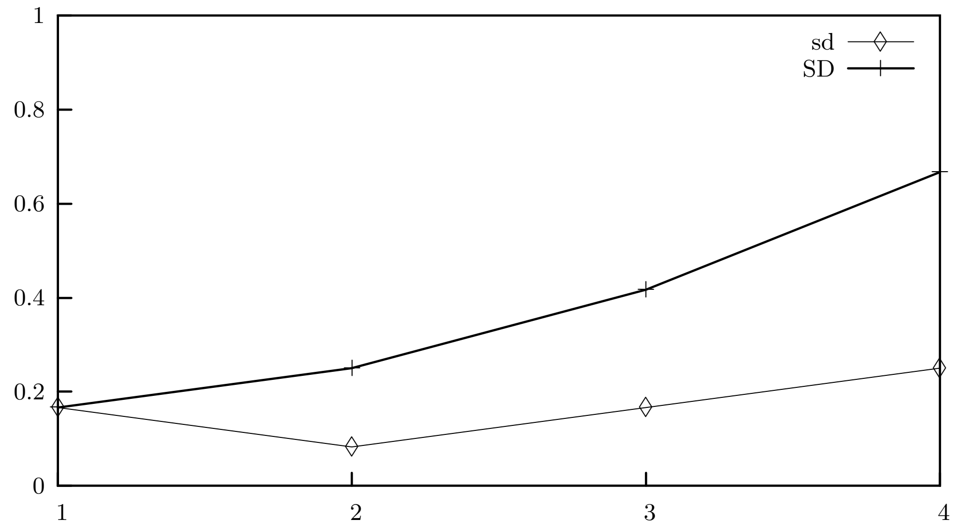


Figure 3.2: A graph of the locality found in the reference stream of Figure 3.1

.

3.4.1 Stack-distance distribution functions

Given a memory-reference trace T consisting of memory references $(t_1, t_2, \ldots t_L)$, define $f(T, i, d)$ to be 1, if i^{th} reference in T has a stack distance of d. Define the functions $sd(T, d)$ and $SD(T, d)$ as follows:

$$sd(T,d) = \frac{\sum_{i=1}^{i=L} f(T,i,d)}{L}$$

$$SD(T,d) = \begin{cases} 1 & \text{if } d = \infty \\ \sum_{i=1}^{i=d} sd(T,i) & \text{else} \end{cases}$$

That is, $sd(T,d)$ is the probability that in the trace T a randomly selected reference will have a stack distance of d, while $SD(T,d)$ is the probability that such a reference will have a stack distance of d or less. Figure 3.2 is a graph of the values of $sd(T,d)$ and $SD(T,d)$ for the trace used in figure 3.1.

3.4.2 Stack distance in the literature

Stack distance, as a means of examining reference streams, has been around for over three decades[49]. In the literature one can find it used as an analytic model of locality [26, 41, 58, 65], as a method to generate artificial traces [6, 8, 33], and as part of an algorithm for hardware simulation [49, 68]. The remainder of this section address is these applications.

Stack distance as an analytic model of locality

It is fair to say that the Stack Distance Model (SDM) was one of the most popular models of cache behavior during the late 1970s [58, 64], slowly replacing the Independent Reference Model (IRM) and the Working Set Model (WSM) [24, 58]. The major difference between the WSM and the SDM is that the WSM counts the number of intervening references, rather than the unique number of intervening references, between successive accesses to the same location. The WSM and the SDM are fairly similar and, for the most part, represent the same information. However, the WSM is both larger to store (its maximum size is bounded by the number of accesses rather than the size of the working set) and is generally more difficult to utilize. A detailed explanation of each of these models, and other related models, can be found in [26].

In 1977, a Stanford technical report was written by Rau on the properties and applications of the stack model [58]. Rau's highly mathematical paper develops a number of important results for the stack distance model when applied to fully-associative caches. Rau also shows that given $sd(T, d)$, one can derive a number of values including the distribution governing how long an item will stay in the cache, some occupancy results results for multi-level caches, and how stack distance values relate to set-associative caches. Most relevant to our work in Chapter 5, Rau finds that if $m_s(q, a)$ is the miss rate for a set-associative cache with q sets of size a, its value can be computed as:

$$m_s(q, a) = 1 - \sum_{i=1}^{N} p(i) B_a(i)$$

These results assume that the cache blocks are randomly mapped into the cache. That is, any given cache block has an equal chance of conflicting with another block. This assumption is empirically demonstrated in [65] as well as in Chapter 5. Also, the realization that hash caches (discussed in section 2.2.1) generally contribute very little in the way of performance gain hints that this assumption is reasonable.

Rau briefly discusses the function $B_a(i)$ and concludes that a more highly set-associative cache will result in a lower miss rate if (but not only if) $\forall i : sd(i) > sd(i + 1)$ is a non-increasing function. In Chapter 5 we re-derive $B_a(i)$ and examine it in much greater detail.

A year after Rau's report, Smith proposed a variation on this caching model [65] and then refined it slightly in 1982 [64]. His and Rau's equations are equivalent, but Smith changes the order of the summations and makes a few other cosmetic changes. Smith is mainly concerned with generating an analytic model for $p(i)$ using the stack distance model, combined with common sense, as a reality check. Smith also provides empirical evidence that the actual miss ratio and predicted miss ratio are very similar. He provides five traces and simply graphs the actual and predicted miss rates and shows that they are very similar. While this evidence is intriguing, [2], among others, criticize this work as not being validated on a wide variety of traces.

Hill and Smith again describe the same analytic model of locality in their 1989 work [41]. This time a much larger set of programs are used to validate the model. They show that the vast majority of programs they test achieve a miss rate within 5% to 10% of the value predicted by the model. These results are more satisfying, but no attempt is made to explain the error or to reduce it.

3.4.3 Algorithmic use of the stack model

The first application of stack distances we are aware of came in 1970. Mattson *et al.* proposed using stack-based algorithms to quickly simulate the hit rate of caches over a wide range of values and replacement policies [49]. Their algorithms apply to only a limited number of replacement algorithms, but those include the LRU, FIFO, optimal, and certain pseudo-random replacement algorithms.

Mattson *et al.* focus on fully-associative caches[5] for nearly all of the paper, but do address maintaining separate stacks for each set in the cache for less-associative caches. The majority of this paper is dedicated to a stack-based optimal placement algorithm [11] and its associated proof of correctness. This paper is interesting because it clearly sparked

[5]It is interesting to note that this paper never actually uses the terms "associativity" or "cache." Rather they use the terms "congruence mapping" and "buffer" respectively. We assume that this is because the terminology used today did not exist or was not standard at that time.

the idea of using a stack-based model as a locality metric as well as a means of artificial trace generation. Further, this paper is the first to propose an efficient method of computing the optimal placement policy of a cache, and would not be surpassed in that respect until work by Sugumar and Abraham [67, 68].

3.4.4 Trace compression

Trace-driven simulation is a commonly used method for evaluating the behavior of a memory hierarchy [75]. The trace is gathered by running the program, either in a simulator or on real hardware, and recording every memory access that occurs. This address trace can then be used to drive a variety of memory hierarchies. Pure simulation, where the processor and memory hierarchy are simulated in tandem, will provide more accurate data than trace-driven simulation. However, trace-driven simulation is considerably faster, and because the future accesses are already known, allows for off-line replacement algorithms to be studied such as OPT.

In 1981 Babaoglu proposed keeping the stack distances found in each trace rather than storing the trace itself [8]. He then proposed generating an 'artificial trace' trace from these values using a Monte-Carlo method. While his technique requires that some number of accesses be wasted warming up the cache, it did allow outstanding savings in terms of disk space. The size of the information stored depends solely upon the largest stack distance considered–it is independent of the size of both the original and generated trace. It would be expected that the same errors encountered by Hill's analytic model [41] would also occur when using Babaoglu's technique.

There exist other stack-distance techniques which allow for perfectly accurate simulations of higher levels of the memory hierarchy. The most recent and probably best of these techniques is used in the area of virtual memory simulations [46]. The basic idea is that if all memory components have at least an associativity of k some portion of the actual memory trace is redundant–mainly those with a stack distance of less than k. They show that for values of $k = 5$ the page-level traces are reduced by a factor of about about 20 on the average. While a minimum associativity of 5 is probably unrealistic for standard cache simulations, it is quite realistic when considering a TLB as the first level cache of the page table.

3.4.5 Conclusion

In this chapter a large number of locality measures were discussed, concluding with an overview of the stack distance measure of locality. Most of these techniques are used to address a specific issue, for example compiler loop optimizations, to understand how locality is filtered by a cache, or to compress memory reference streams. None of these techniques are intended to be a tool for gaining insight into whole-program locality. The models of locality proposed Weikle and Grimsrud come the closest in terms of intent and ability. But both of these work best on the small scale: on a single loop or a small segment of code respectively.

In the following chapter it is argued that the stack distance model is a fine general technique for understanding locality and how caches interact with that locality. Further it can be used to identify excessive conflict, explain the behavior of non-standard caches, and show how caches filter locality out of a reference stream.

Chapter 4

Stack distance as a locality measure

As described in Chapter 1, there is a widening gap between DRAM speeds and processor performance. This widening gap has simultaneously made the interaction between locality and caches more complex as well as more important. The additional complexity is, in part, due to the fairly modern addition of the non-standard cache schemes discussed in Chapter 2. Those caches, while similar to previous cache schemes, none-the-less interact with locality very differently than standard caches. Further, locality itself is modified by the filtering effect of caches. This filtering means that a reference stream to an L1 cache will have significantly different locality properties than the reference stream to an L2 cache [77]. As discussed in Chapter 3 simple rules of thumb such as the *2:1 Cache Rule* and the *three C's* model [23, 39, 41] over-simplify the situation by making assumptions about the locality of reference streams and even about how that locality interacts with the caches. As also discussed in Chapter 3 other models of locality are either too complex [34], too narrow in scope [29, 53], or otherwise inappropriate [76] for considering the locality of an entire program. This whole-program locality is important mainly for cache designers and compiler writers.

The difficulty of understanding whole-program locality makes it very difficult for a cache designer to propose appropriate cache schemes or to understand how and when non-standard cache schemes will provide benefit. Without this understanding the only real options are to rely upon flawed rules of thumb or upon hit-rate results on a large set of simulations. Neither is satisfactory and both can lead to poor designs. Relying upon hit-rate results is problematic because designer needs to worry about a huge set of programs including programs which do not yet exist. This need results in huge amounts of simulation activity as well as significant effort spent on prognostication. A model of locality would, at the least, allow the cache designer to narrow the set of programs used by eliminating those with similar locality properties. The model also could be used to find trends and make more informed predictions about future workloads. That said, the most important use of the model would not be in simply improving current techniques but in allowing the cache designer to better understand locality and how that locality interacts with the various possible cache designs.

It is less obvious that compiler writers also have a need to understand locality at the program level. It is clear and well accepted that "microscopic" locality, as discussed in Chapter 3, is extremely important to complier writers. However, whole-program locality is mostly a function of the program being compiled rather than the compiler itself. Where whole-program locality is important is in the context of data layout. If indeed the whole-program locality is determined by the program and not the compiler the only whole-program issue that can be addressed by the compiler is conflict between the various memory references in the cache. The locality and cache models can be used to predict hit rates given random conflict between references. If, through profiling, it can be determined that a given data layout has conflict that is worse than random, it may be worth having the complier try a different data layout. That is, the cache and locality models can provide guidance as to expected hit rates and this can be tested against the actual hit rates achieved. If the actual hit rate is worse than the predicted value, a new data layout can be proposed and tested. It may be that the program itself has lead to this conflict (for example by striding though a one-dimensional array), but it may be that the arbitrary layout chosen by the complier has lead to 'excessive' conflict.

This chapter provides a unified method for understanding and quantifying locality and how caches interact with that locality. The chapter addresses general concepts of locality (Sections 4.1 and 4.2), interactions with non-standard caches (Section 4.3), as well as applications (Section 4.4) and future work (Section 4.5). Even though there are many details found in this chapter, our primary goal is to provide insight into caching and locality.

4.1 The stack distance model of locality revisited

In this section we describe two models which are based upon the stack distance metric. The first is the locality model, that is, describing the locality of a trace by using the terms $sd(T, d)$ and $SD(T, d)$ as defined in Chapter 3. This model of locality allows for a concise description of the locality of a trace. As is shown later in this chapter, the stack-based model of locality can be used to help understand the degree and type of locality in a trace. Such understanding can otherwise be difficult to achieve when dealing with millions or billions of data references.

The second model is a cache model. It describes how a cache interacts with the locality of a reference stream. Specifically, it predicts the expected hit rate of memory references of a given stack distance. It is important to note that this model is arrived at independent of a specific reference stream: the model describes how the cache can be expected to interact with any reference stream. As is shown later in this chapter, this model can greatly aid in the understanding of caches as well as understanding the real impact of cache structure, associativity and size.

These two models can be used together to produce a number of interesting results. These results include the ability to predict the hit rate a reference stream will achieve on a given cache. None-the-less, it is important to keep in mind that these two models are distinct. The locality model is simply a way of describing the locality of a memory reference stream, while the cache model describes how a cache interacts with the locality of a memory reference stream.

4.1.1 LRU stack distance revisited

In Chapter 3 the idea of stack distance was introduced. In this section that introduction is refined and used to describe the characteristics of the various SPEC CPU2000 benchmarks, hereafter referred to by the name "Spec". These benchmarks are targeted to measuring CPU performance, but are also the standard for measuring L1-cache performance. These benchmarks are designed to mostly fit in the cache hierarchy, and it is well known that these benchmarks do not generally put a significant load on any cache other than the L1 [21].

The Spec benchmarks run for minutes to hours on real hardware and generate well over a hundred billion memory references. Gathering data on such large runs in simulated hardware is extremely time consuming and storing such large traces, even on modern disk drives, is impossible. As such, we limit our studies to samples of one billion memory references are for each benchmark. A reasonable period of time is spent allowing the program to finish its initialization and begin doing actual work (at least 100 million memory references) before this sample is gathered. Unless otherwise noted all traces are gathered from optimized binaries for the DEC Alpha and run on the SimpleScalar simulator. The SimpleScalar website, www.simplescalar.com, has details about this simulator as well as the executables used. These traces were then run though a modified version of the Cheetah simulator [67] to generate the data found in this section. These results were also cross-checked against other, much less efficient, simulators written by this author and others.

Recall from section 3.4 that $SD(T,d)$ is the percent of memory accesses in a memory reference trace T that have a stack distance of d or less. Equivalently, $SD(T,d)$ is the hit rate achieved by a fully-associative cache consisting of $d+1$ cache lines on that same trace. This function is termed the cumulative stack distance of the trace. Figure 4.1 provides a graph of the cumulative stack distance for the first three benchmarks in the floating-point portion of the Spec benchmark. For purposes of readability the X-axis has a logarithmic scale. As will be true throughout this chapter, the cache line size is 32 bytes. Notice that were the cache fully associative, art and ammp both would see very little improvement in hit rate from a size of 50 lines (about 2KB) to 2000 lines (about 65KB). However, ammp would see a huge reduction in its miss rate from 2000 lines to 16000 lines(256KB) while art would see only a minor improvement.

At this time it is worthwhile to emphasize the fact that characterizations can be made about the relative locality of the various traces. For example, figure 4.1 clearly shows that art has less locality than the other two traces. Historically the characterization of a trace was more *ad hoc*. One might examine hit rates over a number of different caches, varying size and associativity. However, in general only "reasonable" caches would be considered—generally powers of two and in the range of 4-64KB in terms of size and usually of small

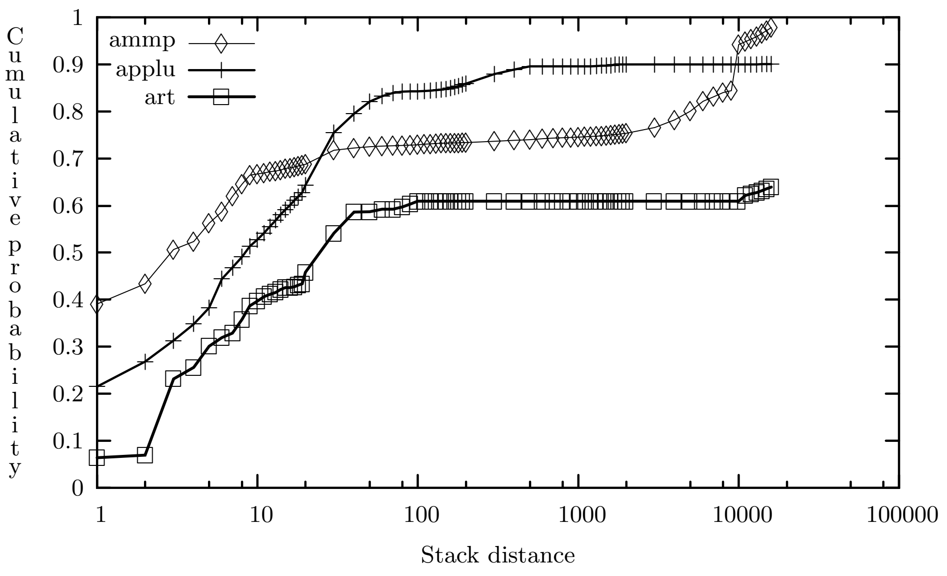


Figure 4.1: Cumulative stack distance for three floating-point benchmark traces

associativity. Instead we can gain detailed insight about the traces by considering a much wider range of sizes. Also, by using fully-associative caches, the impact of conflict between caches of specific sizes does not interfere with our view of the locality inherent in the traces. (In section 4.4.2 it is shown that this allows us to *measure* conflict.)

One final observation is the locality difference between the Spec floating-point benchmarks, which mostly involve scientific applications manipulating large arrays, and the Spec integer benchmarks, which are much more of a mixed bag (compliers, games, graphic applications and more). Intuition is that the floating-point benchmarks operate on larger data sets and thus have less locality. Figure 4.2 shows that this is quite true. SpecFp and SpecInt both follow very similar trends, but SpecFp does have less locality than SpecInt.

4.1.2 Derivation of the cache model for standard caches

As described in Chapter 3, both Rau [58] and Smith [64] have provided equations to predict the miss rate of a cache given the cache's parameters and the probability of an access

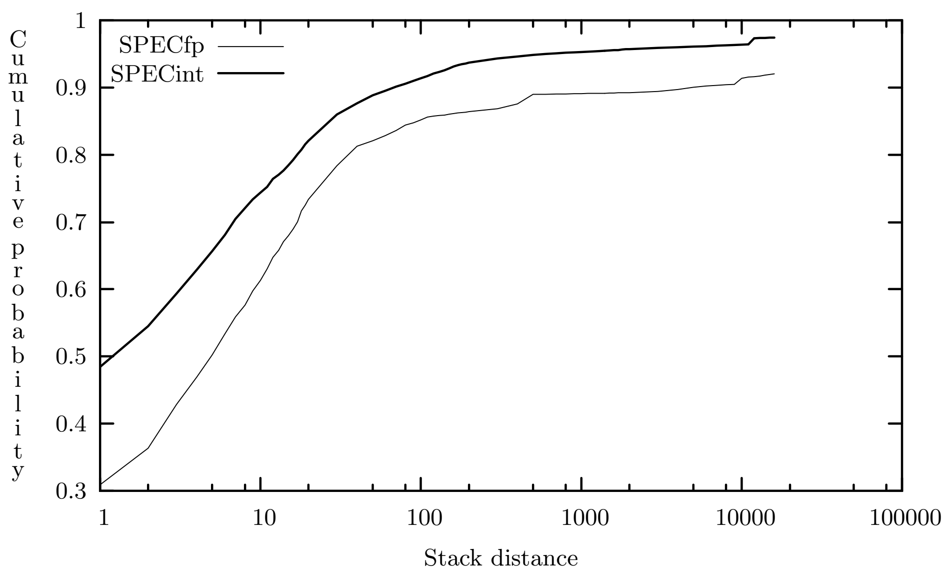


Figure 4.2: Average of the locality found in the SpecInt and SpecFp benchmarks

Symbol	**Meaning**
T	The memory-reference trace
R	The current cache block being analyzed (referenced in T)
E	The set (or line for a DM cache) where R can be placed
D	Stack distance since the last reference to R in T
A	Degree of associativity of the cache
B	Number of lines in the cache
S	Number of sets in the cache (equal to B/A)

Table 4.1: Symbols used in describing how caches interact with locality.

having a certain stack distance. Further, Rau introduced a formulation which allows one to easily separate the cache's characteristics from the program's locality. However, we find their equations somewhat unwieldy, both mathematically and algorithmically. As such we introduce equivalent formulas. In Table 4.1 the various terms used in our formulation are introduced. These terms will be used throughout the remainder of this chapter.

Start by considering the probability that a reference to a given block will be a hit in a direct-mapped cache when there have been D unique intervening accesses since the last reference to that block. The symbols found in Table 4.1 will be heavily utilized throughout this derivation. Once the direct-mapped case is completed, the more general case of an A-way associative cache will then be considered.

Consider a memory reference trace T, which, in addition to other references, includes exactly two references to the block R. In the case of a direct-mapped cache, the second access to R will result in a miss if, and only if, there had been an intervening access to the cache line E since R was last referenced. If $D = 1$, there was exactly one unique intervening access since R was last accessed. Assuming that the intervening access has an equal probability of being stored in any cache line, there is one chance in B that the intervening access was to the cache line E and thus R was evicted.

So the probability of a hit when $D = 1$ is $(1 - 1/B)$ or equivalently $(B - 1)/B$. There are effectively D Bernoulli trials each of which has $1/B$ chance of causing an eviction of R. We want to compute the probability that none of them causes an eviction. The odds of a single intervening access not causing an eviction is $(B - 1)/B$. Thus the odds of none of them causing an eviction is $((B - 1)/B)^D$.

Therefore, the expression $((B - 1)/B)^D$ describes the probability of a direct-mapped cache of size B achieving a hit when there were D unique intervening memory references between the current memory reference to block R and when block R was last referenced. The next step is to generalize this result for an A-way associative cache which uses LRU replacement.

In an A-way set-associative cache there are a number of sets S, equal to B/A. Here a miss will only occur if the number of unique intervening accesses to the set E is greater than or equal to A. So, for a two-way associative cache a hit will occur if there are 0 or 1 such accesses. Using the same argument as in the direct-mapped case, the probability of having none of the accesses go to the set in question is $((S-1)/S)^D$. The probability of having exactly one access to E can be computed using standard counting techniques. $D-1$ of the accesses must map to sets other than E, while one of the accesses must map to E. The probability of $D-1$ accesses using a cache line other than E is $((S-1)/S)^{(D-1)}$. The probability of an access mapping to E is $(1/S)$. Additionally, any of the D accesses could be the one which maps to E, so we need to multiply by D. Thus, for a two-way associative cache we have the following chance of getting a hit:

$$\left(\frac{S-1}{S}\right)^D + D\left(\frac{S-1}{S}\right)^{D-1}\left(\frac{1}{S}\right) \tag{4.1}$$

Notice that this expression is just the probability of a hit on a direct-mapped cache of size B/A plus a term for the second way of associativity. In general, for a cache of associativity A the probability of a hit is equal to the probability of a hit on a cache with associativity $A-1$ plus an additional term. That term will be the probability of exactly $A-1$ of the accesses being mapped to the set E. The general form for that term is:

$$\binom{D}{A-1}\left(\frac{S-1}{S}\right)^{D-(A-1)}\left(\frac{1}{S}\right)^{A-1} \tag{4.2}$$

So for a cache with S sets and associativity equal to A the chance of a hit is:

$$P_{HIT} = \sum_{a=0}^{A-1}\binom{D}{a}\left(\frac{S-1}{S}\right)^{D-a}\left(\frac{1}{S}\right)^{a} \tag{4.3}$$

An equivalent, and sometimes more useful form of the equation results from replacing S with B/A.

$$P_{HIT}(B, A, D) = \sum_{a=0}^{A-1} \binom{D}{a} \left(\frac{B-A}{B}\right)^{D-a} \left(\frac{A}{B}\right)^{a} \tag{4.4}$$

Unless otherwise noted, the formulation of Equation 4.4, as well as the order of arguments, will be used when computing P_{HIT} for standard caches. Figure 4.3 provides a graphical representation of these values for three caches of different associativity, each with 128 cache lines. Notice that the more associative caches do better on references with a low stack distance while the less associative caches do better on data with a higher stack distance. Also note that the crossover point is approximately the same as the number of cache lines.

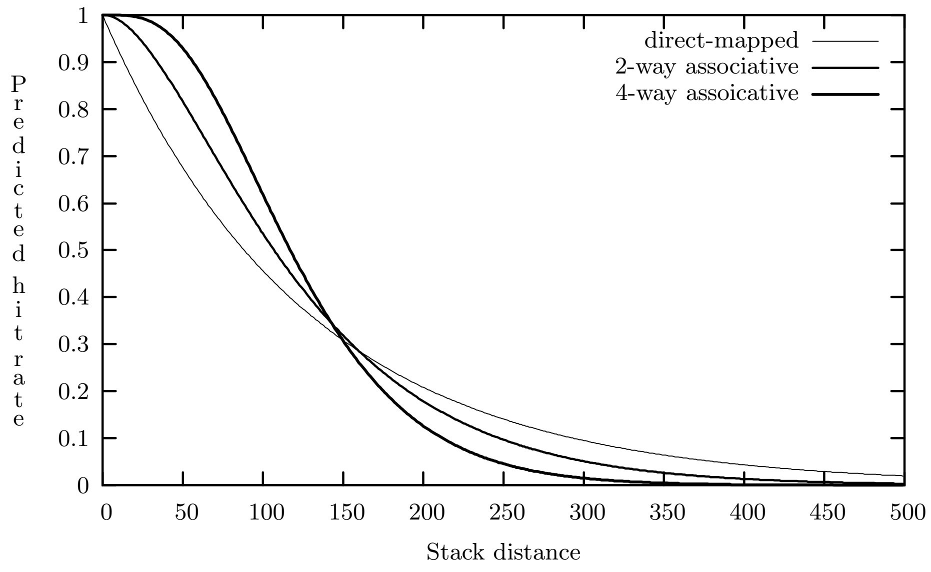


Figure 4.3: Illustration of the theoretic relationship between hit rates, associativity, and the locality of a given memory reference.

A few interesting properties of these formulas are worth mentioning, although formal proofs are not provided. The first is that $\sum_{D=0}^{\infty}(P_{HIT}(B, A, D)) = B$. That is, no matter the associativity, the area under the curve is constant, it is simply distributed at different points. The intuitive argument for this result is that at any given time B blocks are stored in the cache. Say that the status of the cache was sampled N times and the stack distance of each block in the cache was recorded. After a large number of samples, taking the count of each stack distance and dividing it by N should yield the curve described by Equation 4.4. Clearly, as B values are recorded at each step, the summation over all of the stack distances must be B.

Another interesting fact is that for positive integer values of B and D and where the associativity X is greater than Y it is the case that:

$$\sum_{i=0}^{D}(P_{HIT}(B,X,i)) \geq \sum_{i=0}^{D}(P_{HIT}(B,Y,i))$$

That is, if you consider two caches of different associativity but the same number of blocks and sum the area under the curve from zero to some point D the less associative cache will never have a greater sum than the less associative cache. One can observe this phenomenon in Figure 4.3, it is obvious that the less associative cache has more area to the right of any given stack distance. As mentioned in Chapter 3, Rau formally derived a corollary to the above result: that if $sd(T,d) > sd(T,d+1)$ for all values of d, then it is the case that a more associative cache can expect to achieve a higher hit rate.

4.2 Validation of the cache model

The cache model relies upon one fundamental assumption: conflict in a cache is random. This assumption may seem somewhat counter-intuitive for two reasons. First, array-based programs often *stride* though memory at even intervals [39]. This striding can result in excessive conflict in some sets while others are virtually unused. The second reason to cast doubt on the assumption of random conflict is more significant: spatial locality. Recall from

Chapter 2 that standard caches are organized so that two accesses cannot conflict if they are within $S-1$ blocks of each other. For most modern computer systems this means that two cache blocks will never conflict if they are placed into the same page. Because of spatial locality it can reasonably be expected that memory references near each other in time are less likely to conflict with each other than random selection might indicate.

In this section it is demonstrated that while the assumption of random conflict is not exact, it is fairly close. Specifically, we do the following:

- Choose two cache configurations, a 4KB direct-mapped cache and a 4KB two-way associative cache, and show that the predicted hit rates *at each stack distance* map fairly well to the actual hit rates.
- Show that over a wide range of cache configurations the model generally predicts the hit-rate within 2% of actual values.
- Discuss what the error in predicted hit rates *means*.

The last point is expanded upon in Section 4.4.2.

Graphing actual vs. predicted hit rates as a function of stack distance

Figure 4.4 is a graphical representation of the accuracy of our cache model over all the integer benchmarks. The cache in question is a 4KB direct-mapped cache. Obviously the integer benchmarks perform almost exactly as the theory predicts. Accesses with low stack distances seem to do slightly better than predicted. This small difference is actually quite important as three-fourths of all accesses from the SpecInt suite have a stack distance below 10. (You may recall this from Figure 4.2.) None-the-less, the integer benchmarks match theory remarkably well.

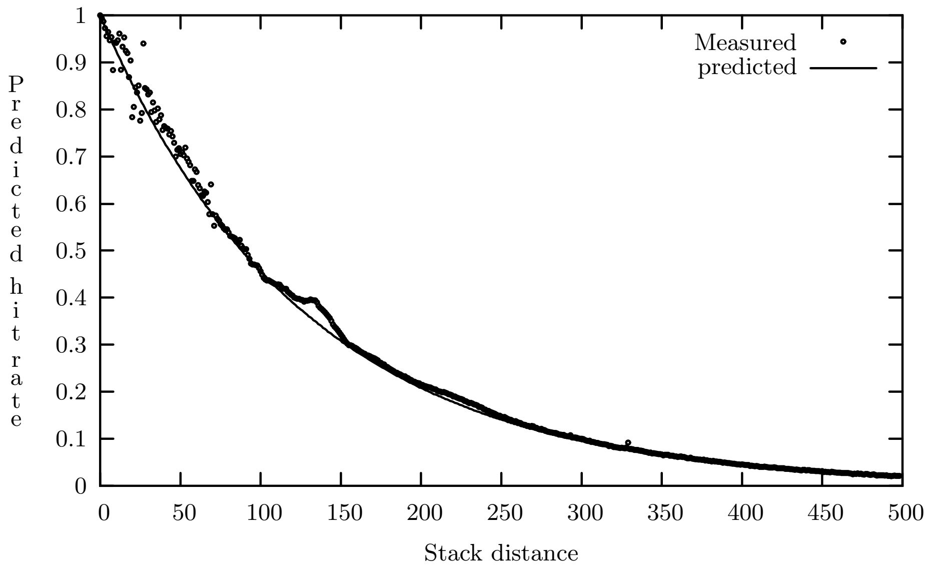


Figure 4.4: Stack distance hit-rates for SpecInt on a 4KB direct-mapped cache.

On the other hand, Figure 4.5 shows that the same data for the floating-point benchmarks is significantly more noisy. While the general trend does track the predicted values (other than perhaps the tail end of the curve) it is not anywhere as close as the integer benchmarks.

This huge difference in locality properties is due to the very nature of the programs. The floating-point benchmarks tend to be fairly small kernels of code working over large data sets. As such the access patterns tend to repeat which reduces the random nature of conflict. The integer programs tend to have larger kernels and do not generally operate on the same data sets over and over again.

Simply to show that the same general results hold for an associative cache, Figures 4.6 and 4.7 provide the same information, for two-way associative caches. The integer results remain very close to the prediction while the floating-point values may be slightly less noisy than they were on the direct-mapped cache.

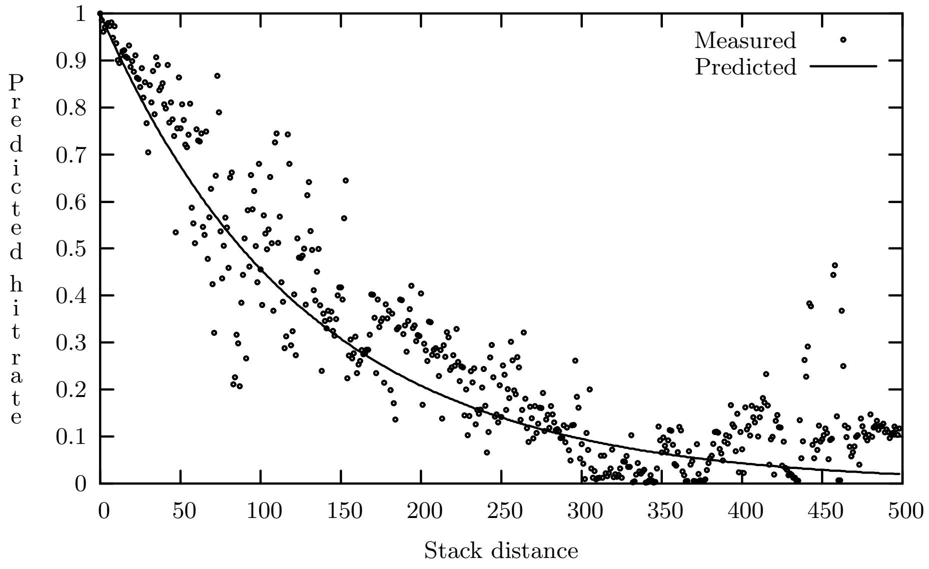


Figure 4.5: Stack distance hit-rates for SpecFP on a 4KB direct-mapped cache.

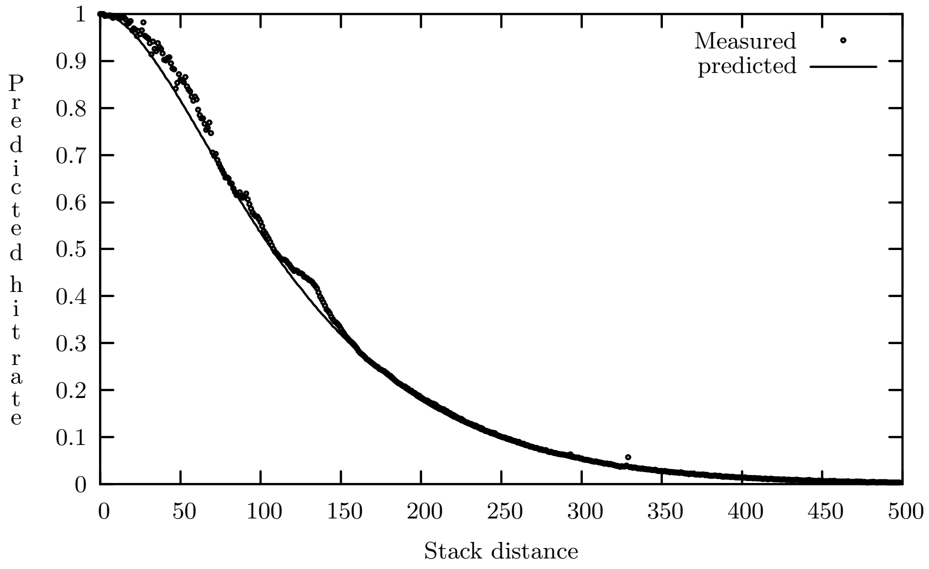


Figure 4.6: Stack distance hit-rates for SpecInt on a 4KB two-way associative cache.

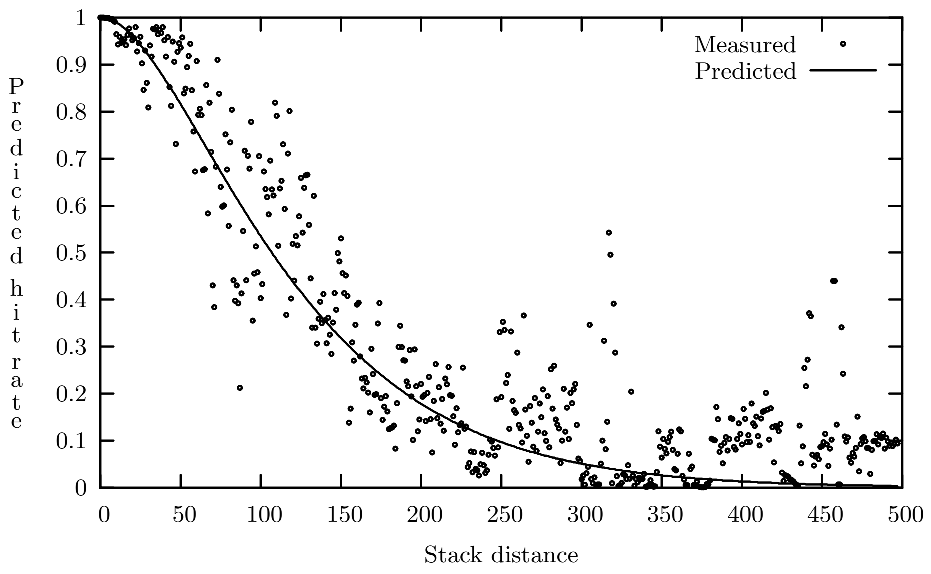


Figure 4.7: Stack distance hit-rates for SpecFP on a 4KB two-way associative cache.

4.2.1 A comparison of actual hit rates to predicted values

Where Section 4.2 illustrated the accuracy of the caching model showing how well it predicted the hit rate at each stack distance over a set of benchmarks, here we measure how well the model predicts actual hit rates. The results are provided by benchmark, cache size, and cache structure. While general trends in accuracy are discussed, the primary goal of this section is to give additional validation for the cache model as well as to provide some insight into what errors should be expected.

Nineteen traces were run on 21 different cache configurations. Those configurations are the cross-product of three cache schemes (direct-mapped, direct-mapped hash, and 2-way set-associative) with seven cache sizes (1KB, 4KB, 8KB, 16KB, 32KB, 64KB and 128KB). All caches have a 32-byte line size. In order to summarize these 399 data points into a useful set of information, tables have been provided which show the average results when two of the axes (benchmark, cache scheme, cache size) are held fixed. The results provided are the average predicted hit rate, the average actual hit rate, the average of the signed errors, and the average absolute error. The floating-point and integer benchmarks are considered separately.

Benchmark	Predicted hit rate	Actual hit rate	Average signed error	Average Absolute error
ammp	71.00%	71.68%	-0.68%	0.68%
applu	82.40%	83.26%	-0.86%	1.27%
art	57.49%	58.48%	-0.99%	1.24%
equake	95.76%	97.21%	-1.45%	1.45%
fma3d	96.63%	96.50%	0.13%	0.87%
galgel	81.41%	82.87%	-1.47%	2.10%
lucas	81.43%	82.58%	-1.15%	1.85%
mesa	94.24%	95.15%	-0.91%	0.99%
mgrid	84.27%	80.31%	3.97%	4.90%
swim	76.35%	79.28%	-2.93%	2.93%
wupwise	94.07%	94.94%	-0.88%	1.00%
Average	**83.19%**	**83.84%**	**-0.66%**	**1.75%**

Table 4.2: Hit rates and predicted hit rates for the SPECfp2000 benchmarks

As can be seen from Table 4.2 the quality of prediction varies quite a bit depending on the benchmark. Three of the benchmarks have errors greater than 1.5%. Recall that if the actual hit rates are better than the predicted values, the conflict in the memory reference stream is lower than it would be if the conflict were purely random. This lower-than-random conflict is generally thought to be a result of spatial locality and the fact that two lines sufficiently close together will not conflict with each other. As discussed at the start of this chapter, poorer hit rates than predicted are due to heavy conflict usually attributable to striding. Notice that mgrid has the largest error and is one of only three benchmarks which does *worse* than predicted. This information shows that the data layout generated by the complier for mgrid has significant room for improvement. More information about the detection of excessive conflict can be found in Section 4.4.2.

Benchmark	Predicted hit rate	Actual hit rate	Average signed error	Average Absolute error
bzip2	94.80%	95.28%	-0.47%	0.57%
crafty	88.21%	88.87%	-0.66%	0.73%
gap	97.93%	98.42%	-0.49%	0.49%
gcc	89.36%	89.96%	-0.60%	0.60%
gzip	94.30%	95.25%	-0.95%	0.95%
mcf	81.09%	82.36%	-1.27%	1.27%
parser	91.61%	92.29%	-0.68%	0.70%
vpr	89.21%	88.60%	0.61%	1.40%
Average	**90.81%**	**91.38%**	**-0.56%**	**0.84%**

Table 4.3: Actual and predicted hit rates for SPECint2000 benchmarks

Examining Table 4.3 the error in hit rate for the integer benchmarks is significantly less than what was found in the floating-point benchmarks. Once again the predicted hit rates are generally a bit lower then reality, while only vpr suffers from more conflict than random.

Configuration	Predicted	Actual	Average signed	Average Absolute
Direct-mapped	82.52%	83.43%	-0.99%	2.00%
Hash	82.52%	83.14%	-0.61%	0.97%
Two-way	84.77%	84.96%	-0.19%	0.47%

Table 4.4: Prediction errors grouped by cache configuration for the SPECfp2000 benchmarks.

Configuration	Predicted	Actual	Average signed	Average Absolute
Direct-mapped	90.20%	90.65%	-0.45%	1.07%
Hash	90.20%	91.06%	-0.85%	0.97%
Two-way	92.04%	92.42%	-0.39%	0.47%

Table 4.5: Prediction errors grouped by cache configuration for the SPECint2000 benchmarks.

One way to think of the cache model is that it takes hit rates for a fully-associative cache and attempts to predict how less associative caches will behave. Unsurprisingly, therefore, Tables 4.5 and 4.4 show that the 2-way associative cache has significantly less error associated with it than the two non-associative caches. What is unusual is that hash caches are supposed to reduce the amount of conflict in a cache without reducing the value of page-level locality. However, on the floating-point benchmarks the hash cache actually harms performance. In any case, the model has the most problems predicting the behavior of the direct-mapped cache. While that error is not trivial, it is plain that the model tracks actual performance fairly well.

Cache Size	Predicted	Actual	Average signed	Average Absolute
1k	73.35%	74.28%	-0.92%	3.22%
4k	84.03%	84.80%	-0.77%	2.15%
8k	86.94%	87.41%	-0.47%	1.25%
16k	88.90%	89.21%	-0.32%	0.81%
32k	90.09%	90.57%	-0.48%	0.60%
64k	90.75%	91.21%	-0.46%	0.55%
128k	91.23%	91.62%	-0.39%	0.56%

Table 4.6: Prediction errors grouped by cache configuration for the SPECint2000 benchmarks.

Finally, Table 4.6 combines the integer and floating-point results as they are impacted by cache size. Clearly the model is much better at predicting the hit rates of larger caches than smaller ones.

From this section a number of things can be concluded. First, the cache model is more accurate when modeling large, associative caches than it is modeling smaller, direct-mapped caches. Secondly, and as noted in [40] the cache model only does a moderate job of actually predicting hit rates. Lastly, and perhaps most importantly, the error in prediction is due to the assumptions made about conflict being random this is not entirely a bad thing. Later in this chapter these errors will be exploited in a novel way to gain additional understanding about caching and conflict.

4.3 Expansion of the model to non-standard caches

It is appropriate at this time to reflect on the stack model as discussed thus far. Recall that in Section 4.1 P_{HIT} for a given distance was shown to be independent of the locality of the trace. In this section it shown that this property does not always hold for non-standard caches. Two non-standard caches are examined: an LRU skew-cache and a true-LRU victim cache. We show that the skew cache cannot accurately be correctly modeled under the assumption that P_{HIT} is independent of $sd(d)$, while true-LRU victim cache can be so modeled.

4.3.1 An analytic LRU skew-cache model

The skew cache was described in Section 2.2.4. An LRU skew cache is simply one where replacement is always to the least-recently used location of those in which the new line can be placed. Building such a cache can be difficult [60], but near approximations are possible [14].

As previously mentioned, a stack-distance based skew-cache model which is independent of $sd(d)$ is not possible. The fundamental reason why is that the model proposed in Section 4.1 relies upon the fact that only the number of unique accesses between successive references to the same cache line are important. That is, it did not matter if those intervening lines were accessed once or 100 times, either way the effect would be the same. With a skew cache, that is not the case. The reason is that if a line is evicted from the skew cache and then re-referenced, it might occupy a different location than it did the first time. This concept, which to our knowledge no one has ever noticed or even had the language to discuss, is best explained by way of an example.

Impact of $sd(d)$ on P_{HIT}

In order to make this example readable, some terminology is introduced. Assume a two-way associative skew cache. The two ways of the skew cache are named A and B. Line i in way A will be called $A[i]$, with a similar notation used for B. Say a given block, Y, can be placed into either the ith line in A or the jth line in B. We use the terminology $Y \rightarrow \{A[i], B[j]\}$ to represent that situation.

Assume that between two successive references to cache line $X \rightarrow \{A[1], B[1]\}$ there are references to three other lines, $L \rightarrow \{A[1], B[2]\}$, $M \rightarrow \{A[2], B[2]\}$, and $N \rightarrow \{A[2], B[2]\}$, as shown in Figure 4.8. Assume that X is initially placed into block $A[1]$. For the moment, assume that M, N, and L are each referenced exactly once. Notice that if L is referenced after M and N have both been referenced, X will be evicted. Further, notice that if L is the first of the three lines (of L, M, and N) referenced, X will not be evicted. If L is the second of these three lines to be referenced, there is a 50% chance X will be evicted (if the previous reference was placed in $B[2]$). Thus, averaging over the six possible orderings of the references to these three lines, there is a 50% chance of X being evicted before it is re-referenced. In this exact situation, we observe that for the second reference to X, $P_{HIT} = 0.5$ and its stack distance is three.

Now, assume that each of these three lines is referenced a large number of times before X is re-referenced. The stack distance for that second reference to X remains the same: three. However, as the number of repetitions goes to infinity, the probability of X being evicted goes to one. The reason for this is that all that has to happen for X to be evicted is that there is *some* reference to L after both M and N are referenced. In that case L will be assigned to $A[1]$, evicting X. In this situation for the second reference to X, $P_{HIT} = 0$ although its stack distance remains three.

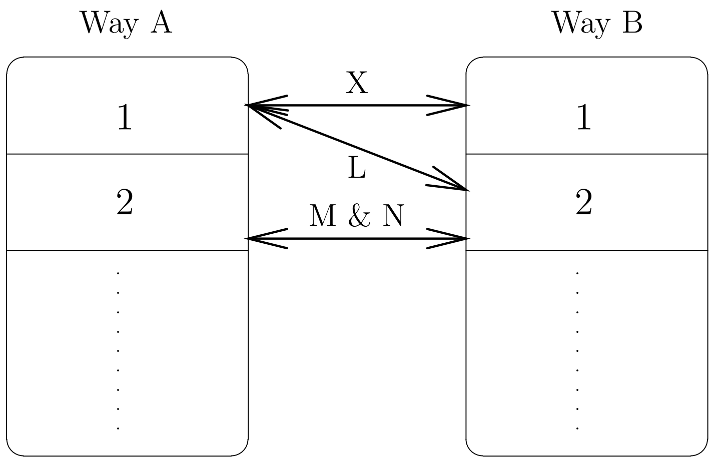


Figure 4.8: Example showing how locality interacts differently with a skew cache than a standard cache.

Thus, without attempting to generalize the effect, it is clear that repeating a reference to the same memory location may not change the stack distance of some other reference, but it can change P_{HIT}. Notice that the distribution of $sd(d)$ plays a role in this. For our example, if $SD(2) = 0$ there will be no repetitions of L, M, or N between the two references to X. If $SD(2) = 0.999$, the probability of repeats will be quite high. Therefore, $sd(d)$ has an impact on P_{HIT}.

Moving forward with the model

The previous example is enough to show that for a skew cache, $sd(d)$ can have an impact on P_{HIT}. While acknowledging that observation, we none-the-less create a model which assumes that P_{HIT} is independent of $sd(d)$. It is then shown that for real memory reference traces that such an assumption only introduces minimal error. *For purposes of this derivation, it is assumed that each intervening memory location is referenced exactly once.*

Our goal is to find the probability that a memory access with stack distance D will achieve a hit. This is equivalent to determining the probability that a given memory reference, R, will remain in the cache after D unique accesses have occurred. For this derivation, assume a two-way associative skew cache where each way of the cache consists of X cache lines. Label the first way A and the second way B. Assume without loss of generality that the reference in question, R, is to a block placed in A. Define a cache block to have been “touched” if one of the D references that occurs after R is placed in that cache block. Define a to be the number of lines in bank A that have been touched since *immediately before* R was referenced. Define b similarly to a but for bank B. Notice that by these definitions, when $D = 0$ it is the case that $a = 1$ and $b = 0$.

Treat the triple (a, b, D) as the state of a cache **where R remains in the cache**. That is, we are only keeping track of the states of the cache where R would still be found in the cache. For a given state, there are at most three states which could have proceeded it. The previous state may have been $(a - 1, b, D - 1)$ in which case the Dth reference mapped to a block in A that had not previously been touched. Similarly the previous state may have been $(a, b - 1, D - 1)$ in which case the most recent reference mapped to a block in B that had not previously been touched. Going from state $(a, b, D - 1)$ to (a, b, D) occurs if the most recent reference mapped to blocks already touched in both A and B but not to the block in A holding R.

The probability of moving from state $(a, b, D - 1)$ to state (a, b, D) is $((a - 1)/X)(b/X)$, which is exactly the probability of mapping to a touched line in A that isn’t R multiplied by the chance of mapping to a touched line in B. Changing state from $(a, b - 1, D - 1)$ to (a, b, D) requires that the most recent reference is to an untouched block in B and a touched block in A, or it was to an untouched block in both A and B but B was the LRU block of the two (which is assumed to happen 50% of the time). The probability of the reference mapping to an untouched block in B is $(X - (b - 1))/X$ while the probability of it mapping to

Transition from	Probability
$(a, b, D-1)$	$\frac{a-1}{X}\frac{b}{X}$
$(a-1, b, D-1)$	$\frac{X-(a-1)}{X}\frac{b}{X}+\frac{1}{2}\frac{X-(a-1)}{X}\frac{X-b}{X}$
$(a, b-1, D-1)$	$\frac{X-(b-1)}{X}\frac{a}{X}+\frac{1}{2}\frac{X-(b-1)}{X}\frac{X-a}{X}$

Table 4.7: Transition probabilities for a two-way associative skew cache.

a touched block in A (including the one holding R) is a/X. The probability of the reference mapping to blocks which are untouched in A or B is $((X-(b-1))/X)\,((X-a)/X)$. In Table 4.7 the probability of each transition is provided.

Let the term $P(a, b, D)$ be the probability of getting to state (a, b, D). Recall that reaching state (a, b, D) implies that R is still in the cache. Using Table 4.7, it is trivial to derive Equation 4.5.

$$P(a,b,D) = \begin{cases} P(a,b,D-1)\left(\frac{a-1}{X}\frac{b}{X}\right) \\ +P(a-1,b,D-1)\left(\frac{X-(a-1)}{X}\frac{b}{X}+\frac{1}{2}\frac{X-(a-1)}{X}\frac{X-b}{X}\right) \\ +P(a,b-1,D-1)\left(\frac{X-(b-1)}{X}\frac{a}{X}+\frac{1}{2}\frac{X-(b-1)}{X}\frac{X-a}{X}\right) \end{cases} \tag{4.5}$$

The initial condition is that when $D = 0$, $a = 1$ and $b = 0$. So $P(a, b, 0) = 1$ if $a = 1$ and $b = 0$, otherwise $P(a, b, 0) = 0$. Now define $P_{HIT}^{Skew}(X, D)$ to be the probability of a skew cache with two banks of size X to achieve a hit on access with a stack distance of D.

$$P_{HIT}^{Skew}(X,D) = \sum_{i=0}^{X}\sum_{j=0}^{X} P(i,j,D) \tag{4.6}$$

As Equation 4.6 indicates the probability of a hit occurring at a stack distance D is simply the probability of reaching a state $(-, -, D)$ where the "–" indicates any value. Figure 4.9 shows the values of $P_{HIT}^{Skew}(64, D)$, that is the chance of a hit occurring in a skew cache of 128 cache lines at a given stack distance. P_{HIT} for a two-way associative standard cache of the same size is also displayed to provide some context. Notice that the skew cache is slightly "more associative" than the 2-way associative cache.

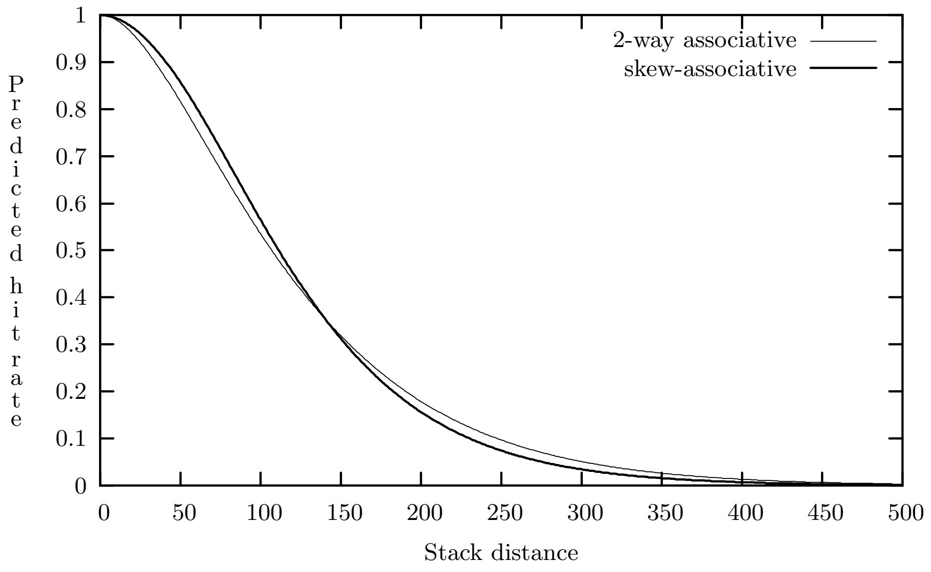


Figure 4.9: Predicted hit rate for a skew cache and a standard 2-way associative cache. Both are 128 lines in size.

A more real situation

As noted in Section 4.3.1, P_{HIT}^{Skew} does depend upon $sd(d)$. Using simulation, one can show the relationship between our predicted value of P_{HIT}^{Skew} and how it is impacted by $sd(d)$. For most realistic $sd(d)$ distributions, our prediction is fairly close to reality. In Figure 4.10 the P_{HIT} is graphed for the analytic model as well as two different $sd(d)$ distributions: that of the integer Spec (as seen in Figure 4.2) and a uniform (flat) distribution where the first 300 stack distances each have a 0.3% chance of occurring. It is worth emphasizing that these graphs are not of actual traces. Rather they are traces which have been artificially generated to have the appropriate $sd(d)$ distributions (see Section 4.4.1 for details). For a standard cache there would be no significant variation between these three lines.

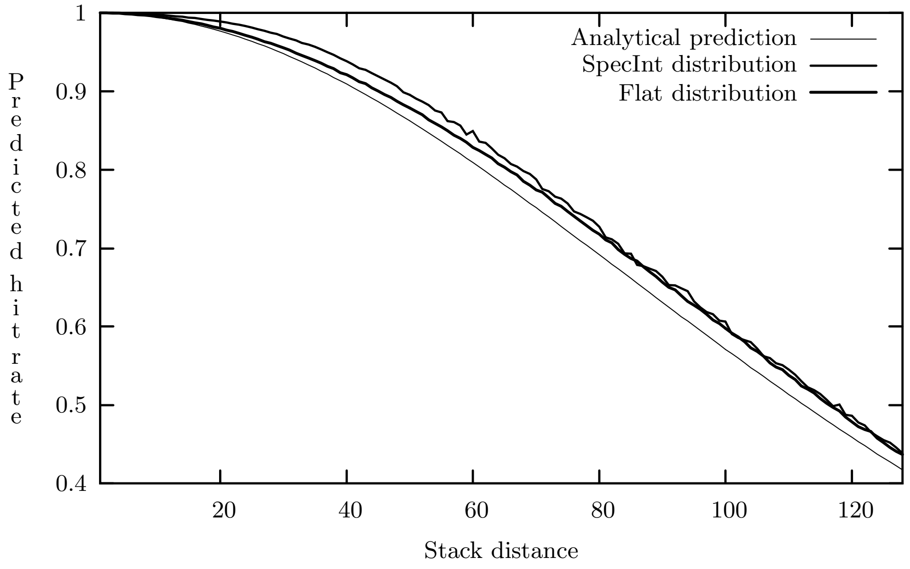


Figure 4.10: A magnified look at the 'flaws' in the skew cache analytic model.

Another issue with respect to the skew cache model involves the assumption that if an access maps to untouched blocks in A and B,, it has a 50% chance of going to either of them. In fact, assuming the first block maps to a location in A means we learned a little bit about A and B—there is at least one location in B which is newer than the line we replaced in A. This leads to a small bias. In Section 4.3.3 we show empirically that providing a 0.501 chance of the line being mapped to B in a 128-line skew cache significantly reduces this already very small effect.

4.3.2 An analytic LRU victim cache model

The victim cache, described in Section 2.2.3, is another type of non-standard cache. Briefly, it consists of two parts; a large main cache, and a much smaller fully-associative component[1].

[1]It is more common to refer to the smaller fully-associative component as the victim cache. However, for reasons of convenience and clarity, the term 'victim cache' is used here to describe the two components together, functioning as a whole.

It is usually the case that the large main cache is direct-mapped to reduce access time. Lines evicted from the direct-mapped component are placed in the fully-associative component which is managed like a separate cache.

A 'true-LRU' victim cache is identical to the usual victim cache except that the fully-associative component is managed using global LRU information. That is, when a block gets evicted from the direct-mapped component, its "time stamp" is compared to those blocks in the fully-associative component. The LRU block is evicted from the victim cache. If the evicted line was in the fully-associative component, the line currently in the direct-mapped component is moved into that location. In any case the direct-mapped component is now unoccupied, leaving room for the block which caused the eviction. In the remainder of this section we provide an analytic model which describes how a true-LRU victim cache interacts with the locality of a reference stream.

Derivation of the model

To facilitate this derivation, the following terminology is introduced. The direct-mapped and fully-associative component are said to contain a total of S_1 and S_2 cache lines respectively. Let the direct-mapped component consist of cache lines labeled as $l_1, l_2, \dots l_{S_1}$. Further, if X is a cache block which maps to location l_x in the component, let this be stated as $X \in l_x$.

As before the goal is to compute the probability that a given cache block will still be in the cache after D unique accesses have occurred. The fundamental observation is that an LRU victim cache will evict a block, X, in cache line l_x from the combined components if, and only if, the following two conditions are met:

- Some other block is accessed after X and is also mapped into l_x.

- At least S_2 other blocks have been accessed and then evicted from the direct-mapped component since X was last accessed.

The first condition must be true, otherwise X will still be in the direct-mapped component. The second must be true, otherwise X can be in the fully-associative component. Notice that the order in which these two conditions occur is not important. Once both have occurred X will be evicted from the victim cache.

Say that there are exactly D unique memory blocks accessed between two accesses to block X. In order to meet the first condition, at least one of those D blocks must be mapped into l_x. Meeting the second condition requires that no more than $D - S_2$ different cache lines are utilized by those D accesses. That is because only those lines accessed *after* X can contribute to the S_2 blocks needed to evict X from the fully-associative component of the LRU victim cache.

Given the above result, it is possible to construct a relationship which computes the probability of a given set of accesses accessing no more than $D - S_2$ different cache lines. The same assumption of "random conflict" used before is used once again.

The number of times an access to a given block repeats has no influence upon the probability of X being evicted. Consider the possibilities if one of the D blocks is accessed repeatedly. Label the block which is repeatedly accessed as R. Upon the repeated access, R is currently in one of three places: the direct-mapped component, the fully-associative component, or has been evicted from the cache.

- If R is in the direct-mapped component, the only change is to update the access time of R. As it was already more recently used than X this action has no effect on X.

- If R is in the fully-associative component, some other of the D accesses must have evicted it from the direct-mapped component. R will be placed in the direct-mapped component while the other block will either go into the fully-associative component or be evicted. Both of these blocks are more recently used than X, so the swapping has no impact on whether or not X will be evicted.

- If R has been evicted, and X is still in the LRU victim cache, it must be the case that X is in the direct-mapped component. Otherwise X would have been evicted from the fully-associative component before R. At this point X will be evicted from the entire victim cache once an access occurs that uses l_x. As R clearly does not map to l_x (otherwise X could not be in the direct-mapped component) repeated accesses to R have no impact.

Thus, the number of times any of the D accesses repeat has no influence upon the probability of X being evicted.

The analytic model

Define $P_{HIT}^{LRUvc}(S_1, S_2, D)$ to be the probability that an LRU victim cache with a direct-mapped component of size S_1 and fully-associative component of size S_2 will achieve a hit on a access with a stack distance of D.

For a fixed value of S_1 and S_2 let $F(D, U)$ be the probability that at stack distance of D exactly U different cache lines have been accessed, *none* of which are l_x. For the same fixed value of S_1 and S_2, let $E(D, U)$ be the probability that at stack distance of D exactly U different cache lines have been accessed, one of which *is* l_x. It then follows that:

$$P_{HIT}^{LRUvc}(S_1, S_2, D) = \sum_{U=0}^{D} F(D, U) + \sum_{U=D-S_2+1}^{D} E(D, U) \tag{4.7}$$

The first summation of Equation 4.7 represents the probability that X is still in the direct-mapped component of the cache. The second summation in that equation represents the probability the X is in the fully-associative component of the cache. For that second summation, notice that this is the probability that l_x has been accessed (and thus X is not in the direct-mapped component) plus the probability that enough different cache lines have been accessed ($D - S_2 + 1$ or more) that X will still be in the fully-associative component of the LRU victim cache.

All that is left is to compute the various values of F and E. It is convenient at this time to define the state of the LRU victim cache as the triple (G, D, U). D and U are defined as above, while $G = 0$ if l_x has not been accessed since X, and $G = 1$ otherwise. This state will be used to create a recurrence relation as was done for the skew cache.

We now define the initial conditions. Notice that if $D = 0$ then $U = 0$. Therefore, $F(0, 0) = 1$, while $F(0, i) = 0$ if $i \neq 0$. Also note that if $D = 0$ then l_x could not have been accessed after X (as nothing has been) and therefore $E(0, i) = 0$ for all i.

The recurrence relationship for F is now defined. In order to reach the state $(0, D, U)$, the previous state had to be either $(0, D - 1, U - 1)$ or $(0, D - 1, U)$. A transition from $(0, D - 1, U - 1)$ to $(0, D, U)$ will occur if the most recent access went to a cache line that has not been accessed since X was placed in the cache and that newly accessed cache line is not l_x. This will happen with probability $(S_1 - (U + 1))/S_1$. A transition from $(0, D - 1, U)$ to $(0, D, U)$ will occur if the most recent access went to a cache line that has been accessed since X was placed in the cache. This will happen with probability U/S_1. From these observations the following recurrence for F can be trivially derived:

$$F(D, U) = F(D - 1, U - 1) * \frac{S_1 - (U + 1)}{S_1} + F(D - 1, U) * \frac{U}{S_1} \tag{4.8}$$

The recurrence relationship for E can be derived in a similar way. In fact the only difference is the case when the most recent access used l_x, and l_x had not otherwise been used after X was placed in the cache. Obviously, this transition will occur with probability $(1/S_1)$. Thus the recurrence relation for E can be expressed as:

$$E(D,U) = \begin{cases} \left(E(D-1,U-1) * \frac{S_1-(U+1)}{S_1}\right) + \left(E(D-1,U) * \frac{U}{S_1}\right) \\ + \left(F(D-1,U-1) * \frac{1}{S_1}\right) \end{cases} \tag{4.9}$$

Equations 4.7, 4.8 and 4.9 are enough to compute $P_{HIT}^{LRUvc}(S_1, S_2, D)$. In Figures 4.11 and 4.12 the P_{HIT} values at various stack distances are plotted for a an LRU victim cache with $S_1 = 128$ and $S_2 = 6$. The hit rate achieved by each component is also graphed separately. The line labeled "victim cache" is the sum of the hit rates of the two components. Notice that the direct-mapped component of the LRU victim cache behaves exactly the same as a direct-mapped cache of the same size. Further, notice that unlike simply increasing the associativity of the cache, the hit rate at low stack distances is dramatically improving without any harm to the expected hit rate at larger stack distances.

4.3.3 Verification and the 100% miss-rate model

There are also empirical methods for finding P_{HIT} for any arbitrary caching scheme. The most obvious method is to build a cache simulator and feed it data accesses, keeping track of the hit rates for the various stack distances, much as was done in Section 4.2. However, doing this is not as simple as it appears, even using the artificial trace generation techniques discussed in Section 4.4.1.

The major difficulty with the above "nïave" simulation method is that some distribution of stack distances will need to be selected. Because P_{HIT} varies with $sd(d)$ for some caching schemes, one would ideally use the same $sd(d)$ distribution for all cache schemes. However,

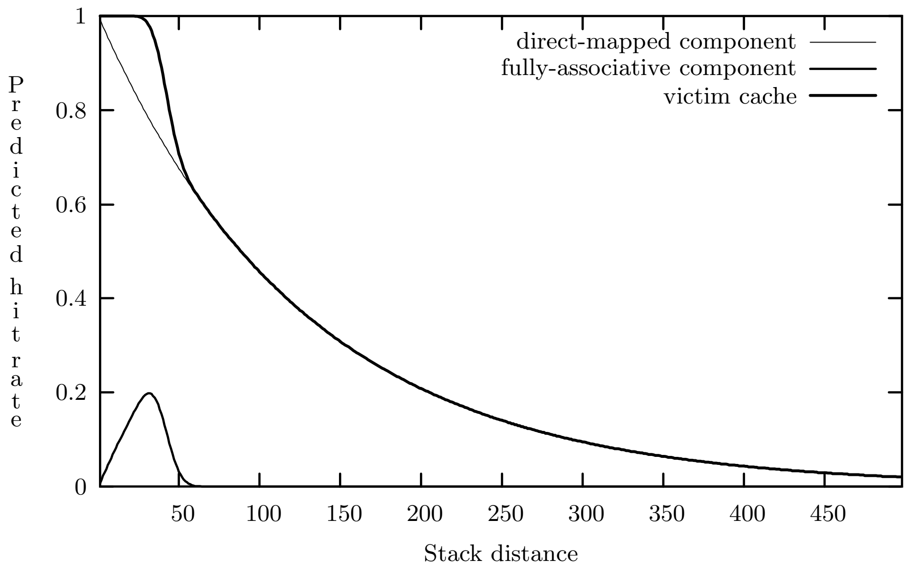


Figure 4.11: Predicted hit rate at each stack distance for the LRU victim cache and its components.

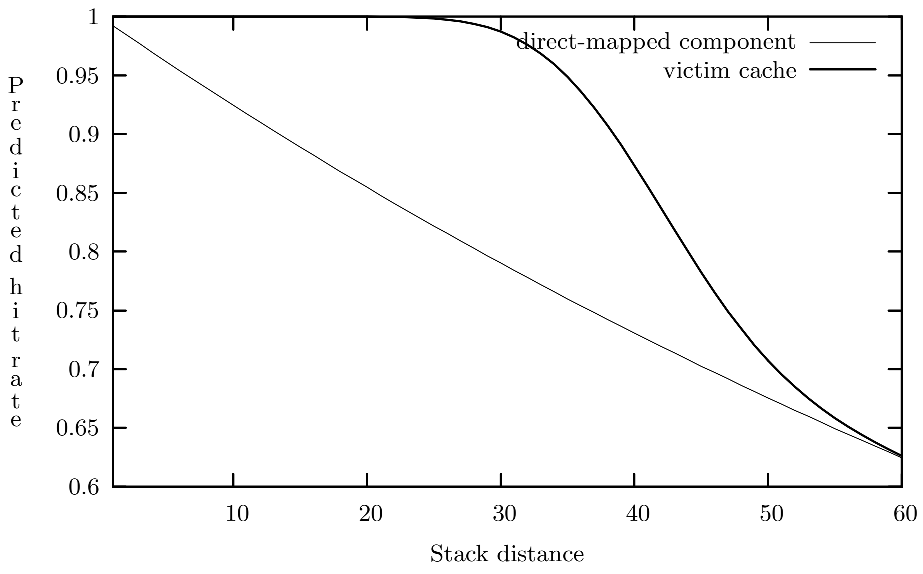


Figure 4.12: A magnified version of the upper-left hand corner of Figure 4.11

if that distribution includes large stack distances for a small cache, time will be wasted collecting unneeded data. For large caches, the opposite problem might occur where the distribution does not include some stack distances of interest. Another very noticeable problem is the time required to run the simulation and gather the needed values. Simple statistical analysis indicates that about 100,000,000 samples are required at each stack distance to get the standard deviation for the estimate of the $P_{HIT}(d)$ below 0.0001. When thousands of different stack distances are being examined, the run time to gather this data can be on the order of hours or days.

Instead a slightly different technique is used. The reference trace used is a random one made up of non-repeating random addresses. That non-repeating stream is identical to a reference stream generated by the techniques discussed in Section 4.4.1 where $sd(\infty) = 1$ and $sd(d) = 0$ for all finite values of d. Rather than keeping track of the hit rate for each reference, the stack distances kept in the cache after each step are recorded. This distribution of locations in the cache indicates which stack distances *would* result in a cache hit if it were the location next accessed. We have named this analytic technique the "100% miss rate model." It allows us to use the same $sd(d)$ distribution for every cache studied. Further, one can trivially modify this technique to find $P_{HIT}(d)$ for any $sd(d)$ distribution.

In order to show that the 100% miss rate model is accurate, it can be compared to the analytic model of the same cache scheme. Miss rates were gathered using 1,000,000,000 accesses to a true-LRU victim cache consisting of a 128 line direct-mapped component and a four line, fully-associative, victim buffer. If each of the billion accesses were treated as an independent test for a given stack distance, it would be expected that the error would be under 0.00003 for the vast majority of the accesses[2]. As can be seen in Figure 4.13 the

[2]The expected error varies somewhat with the expected miss-rate. The 0.00003 value is for the worst-case of 0.5 where it provides approximately a 95% confidence interval

error rate falls well within that range.

A similar graph is presented for a 2-way associative skew cache with a total of 128 lines. As discussed at the end of Section 4.3.1, there is a small error in the analytic model. That error is due to the fact that when a line is placed in a given bank due to an LRU decision, a very small amount of information is gained about the state of the cache. The error creates a small bias which can be greatly reduced with a small change. Figure 4.14 shows the error in prediction with the bias set to .5/.5 when selecting a block to evict when both potential blocks had been 'untouched' up to that point. The measured errors are noticeably worse than randomness would predict (though still very small). However, changing the bias to 0.501/0.499 brings the error into the expected range of 95% of the data having an absolute value of less than 0.00003.

While the error in the analytical model is very small, the 100% miss rate model allows the error to be detected. If the miss rate were anything other than 100% it would be difficult, if not impossible, to know if the locality of the input were causing this small error (as a skew cache's performance is improved by locality).

4.4 Applications of the Models

As a conceptual tool, it is difficult to overstate the value of the model of locality presented in this chapter. Such a tool makes it possible to see and understand the amount and nature of locality in a given trace. The cache models which were introduced in this chapter allow for an understanding not only of how caches interact with locality, but provide a straightforward method of comparing the effects of caches.

In this section applications of the two models are provided. In many cases these applications simply use the models to aid in understanding a given situation or dynamic. In other

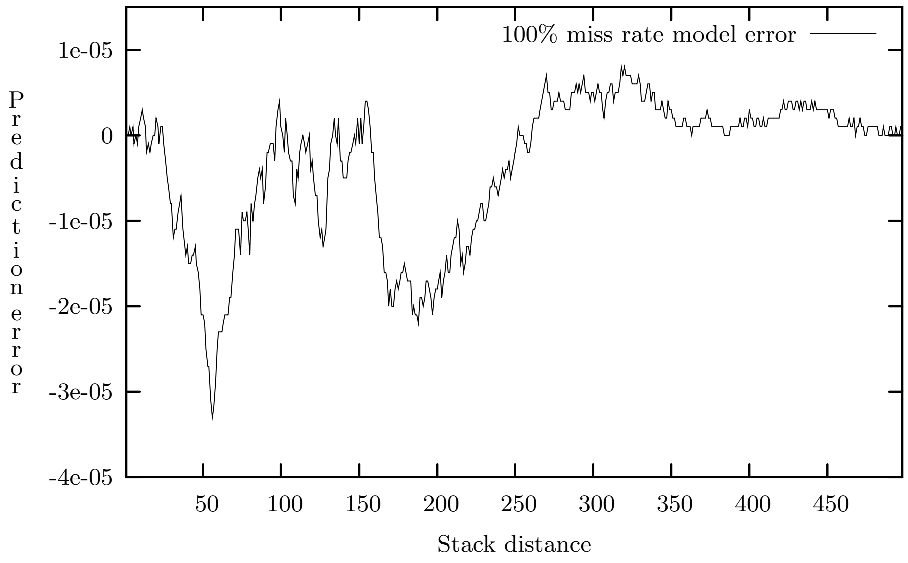


Figure 4.13: Error rate of the 100% miss rate model when predicting the hit rates for a victim cache.

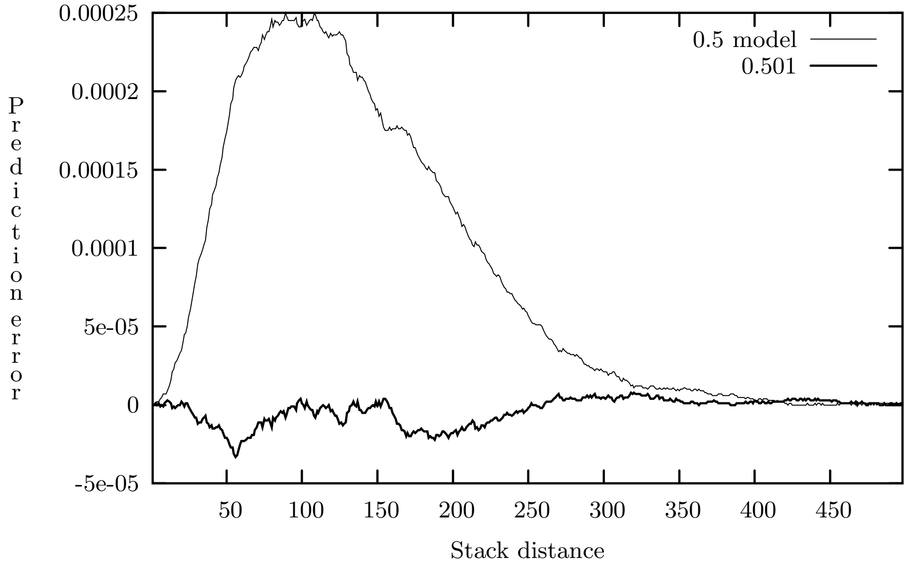


Figure 4.14: Difference between the skew cache analytic model and the same cache using the 100% missrate model. Two different analytic models are compared.

cases these applications allow comparisons and analysis to occur which would otherwise be difficult or impossible. Applications of the locality model are discussed in Section 4.4.1. In Section 4.4.2 applications of the cache model are provided, while Section 4.4.3 provides applications that involve both models.

4.4.1 Applications of the locality model

Figure 4.2 provides what is perhaps the most important insight into locality found in this chapter—it illustrates the nature of locality in the Spec benchmark suite. It shows that two-thirds of all memory accesses would hit on a 512-byte, fully associative cache. Further, it shows that there is not a simple logarithmic relationship that describes locality–there is a highly noticeable knee in the curve near a stack-distance of 50. This observation contradicts the model proposed by Chow [20] (discussed in Section 3.2.1) and is closer to the two-part model of locality discussed Section 3.2.2. Of course the Spec benchmark suite is by no means representative of all workloads. None-the-less, most cache studies focus on these benchmarks and understanding them is quite important.

Beyond simply understanding the whole-program locality of a benchmark suite, the locality model can also provide insight into other issues. In the remainder of this section two applications are examined. The first looks at how locality is filtered by L1 caches. The second discusses how the locality model can be used to generate artificial traces.

Locality filtering by caches

One interesting question is how locality is effected by caches. That is, caches tend to get hits on high-locality memory accesses. Thus, the reference stream of misses and write-backs issued *from* the cache seems certain to have less locality than the reference stream going *into* the cache. As noted in Section 4.4.1, Weikle and her co-authors attacked the issue of measuring how caches filter "instantaneous" locality [77]. As expected, they found significantly less locality in the reference stream once filtered by the cache. However, it is very difficult to get a feeling for the locality of even a moderate sized data stream with their scheme. (Recall that the largest set of memory accesses examined using their scheme was under 4,000 accesses.)

Figure 4.15 shows the impact of locality filtering by a 64KB direct-mapped cache on the gcc benchmark. That is, it illustrates the locality of the reference stream as it leaves the L1 cache and enters the L2 cache. The graph is interesting for a number of reasons. Notice that two different filtered results have been provided in addition to the unfiltered locality information. The 'all reads' line treats all reads and writes to the L1 cache as if they were reads, while the 'read/write' line treats writes as normal in a write-back cache. The difference between the two lines represents the impact of write-back accesses.

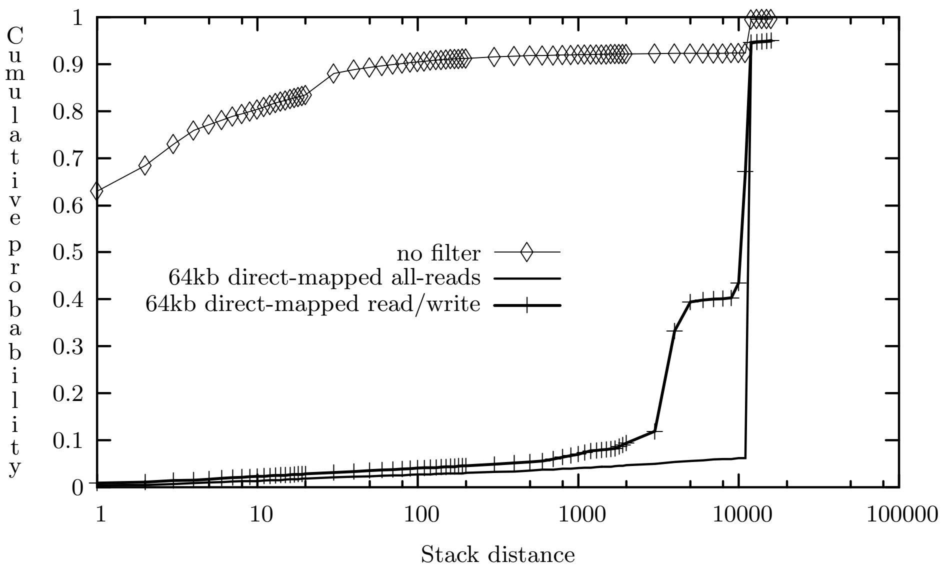


Figure 4.15: Impact of cache filtering on the gcc benchmark and the impact of writebacks

First, notice that the unfiltered trace is very flat from stack distance 100 to about 10,000. As the 64KB cache will get a hit on a stack distance of 100 over 90% of the time, the vast majority of the accesses with a stack distance less than 100 will be hits. As expected, ignoring the impact of write-backs, this characteristic results in having very few (fewer than 10%) of all accesses emerging from the L1 having a stack distance of less than 10,000. The filtered trace then rises dramatically at a stack distance at just over 11,000, a magnified version of a similar jump in the unfiltered trace.

Secondly, the write-backs are interesting. They clearly add a fair bit of locality to the filtered trace, specifically accounting for about one-half of the accesses with a stack-distance less than 2000 (though the total number of access at 10,000 or less is still very near 10%.) They seem to level out at around a stack distance of 5000. The cumulative probability only increases again due to the read misses.

Perhaps the most important point about this data is the huge difference in locality between the filtered and unfiltered traces. In the unfiltered trace a 16-entry fully-associative cache would get a hit-rate of about 90%. In the filtered trace it is less than 3%. The entire premise of caching is that high locality accesses are more common than low-locality accesses. However, after the L1 filtering effect this is not true at all. *The fundamental point is that L1 and L2 caches see extremely different locality properties.* Admittedly this instance is a bit biased. The L1 cache is relatively large (though certainly reasonable in size). Figure 4.16 shows how smaller caches filter locality (in this case all accesses were treated as reads.) It also shows the impact of having a more associative cache.

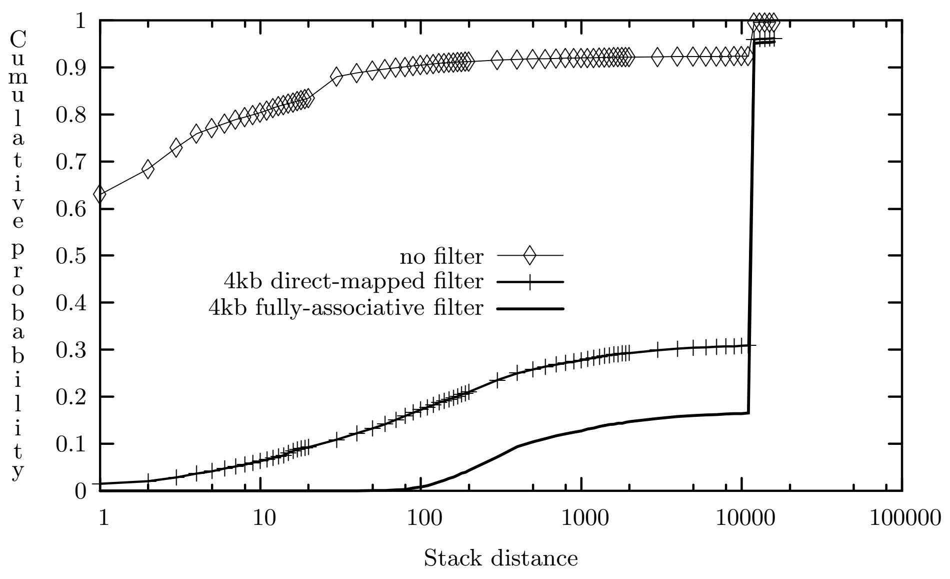


Figure 4.16: Impact of associativity on cache filtering

Generation of artificial traces

Another application of the locality model is its ability to generate artificial traces using a simple Monte-Carlo technique. This was briefly described in Section 3.4.4, and is described in detail in a paper by Babaoglu [8]. The net effect is that it is trivial to generate a memory reference trace which has any stack-distance properties desired. Babaoglu's concern was with a lack of storage space needed to keep long traces. While this is still an issue, disk drive space has been outpacing CPU power for many years; and the bound on trace driven simulation is now mainly CPU power and the network bandwidth needed to move these large traces around. Further, these artificially generated traces are not perfect replicas of the traces they are replacing. For one thing, they have truly random conflict. On these traces a hash-cache will, on the average, perform exactly as well as a standard cache.

Even given the above limitations, these artificially generated traces have many interesting uses. For one, it is possible to generate a reference trace that has certain specific locality properties. This feature can be useful when, for example, you want to know how a change in the locality stream presented to the L1 cache will impact the reference stream seen by the L2 cache. Mathematically analyzing this problem is very difficult, and searching for a real trace that has the locality properties in question will be time consuming and may not yield a match.

Another use of an artificially generated trace is as a point of comparison to a real trace. For example, Hallnor and Reinhardt [36] have proposed a replacement scheme which relies on the idea that past heavy utilization of a cache line will likely result in future heavy utilization. For the trace generated by the Monte-Carlo method the degree of past utilization has nothing to do with the future utilization. The characteristics of the actual trace can then be compared with the characteristics of the artificial trace.

We define a *streak* of memory accesses to be the list of accesses in a given trace which fit in a given cache line. The length of a given streak is defined to be the number of memory accesses in that streak. Figure 4.17 provides a graphical representation of what percent of memory access are in a streak of length N or more for both an actual trace (the first 10 million accesses in the gcc trace) as well as for a similarly sized artificial trace generated using the same stack distances as the original trace. As can be seen accesses in the actual trace are *much* more likely to be a part of a long streak than are those of the artificial trace.

Certainly an empirical model of expected streak length could be generated and used instead of the artificial trace, though doing so seems fairly difficult. Instead we showed that Hallnor and Reinhardt's assumption about heavy utilization being predictive of future utilization quite accurate for this trace. Further, artificial traces can often be used as a point of comparison when there is a desire to separate out the impact of locality from other trace characteristics. For example, it can be used to eliminate the impact of hashing (and

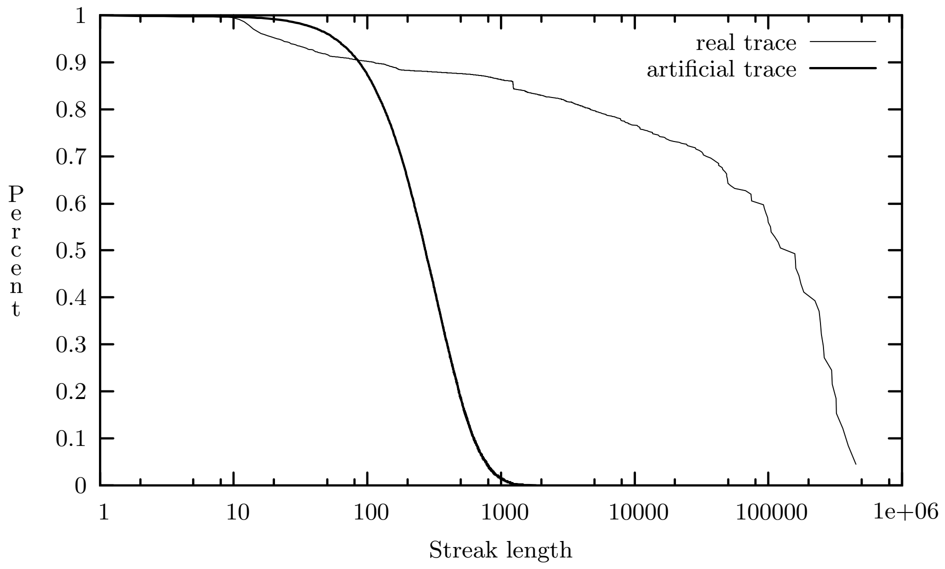


Figure 4.17: Percent of accesses in a streak of size N or greater.

thus the reduction in non-random conflict) while keeping the locality of a given reference stream. This would allow for a measurement of the *reorganization* ability of a skew cache [60].

4.4.2 Applications of the cache model

The cache model also can be used to gain insight into the behavior of caches and reference streams. In this section some of those insights are explored.

Associativity vs. cache size

The inside cover of the most widely used computer architecture text has provides the following under "Rules of Thumb" "**2:1 Cache Rule:** *The miss rate of a direct-mapped cache of size N is about the same as a two-way set-associative cache of size $N/2$.*" Figure 4.18 shows the expected hit rates for $N = 128$. If the rule of thumb is true and one ignores excessive conflict (and as was seen in Section 4.2 there is actually *less* conflict than random for most traces so this is very reasonable), it must be that there are so many memory references in the range of 1 to 23 that the small advantage the two-way associative cache has in that range is enough to compensate for the huge difference in hit rates after stack distance 23. Figure 4.19 magnifies this portion of the graph. The “area” of that small region is slightly larger than 0.6. While the area between the two regions starting at stack distance 24 and onwards is slightly more than 64.6.

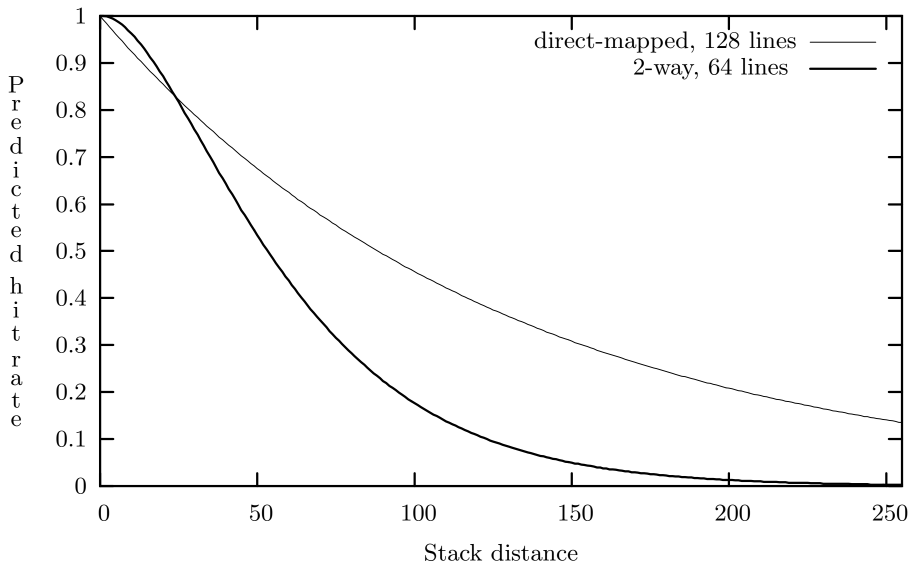


Figure 4.18: Impact of associativity on cache filtering

None of the above analysis should be too surprising, after all Figure 4.2 showed that the vast majority of accesses have a very low stack distance. What is interesting is that the 2:1 Cache Rule is really a statement about the locality of most reference streams (a large percentage of memory references have very low stack distances) rather than about associativity or cache size.

The point of this discussion is not the validity of the rule of thumb. Rather it is that using our model of cache behavior the underpinnings of the rule of thumb can be understood.

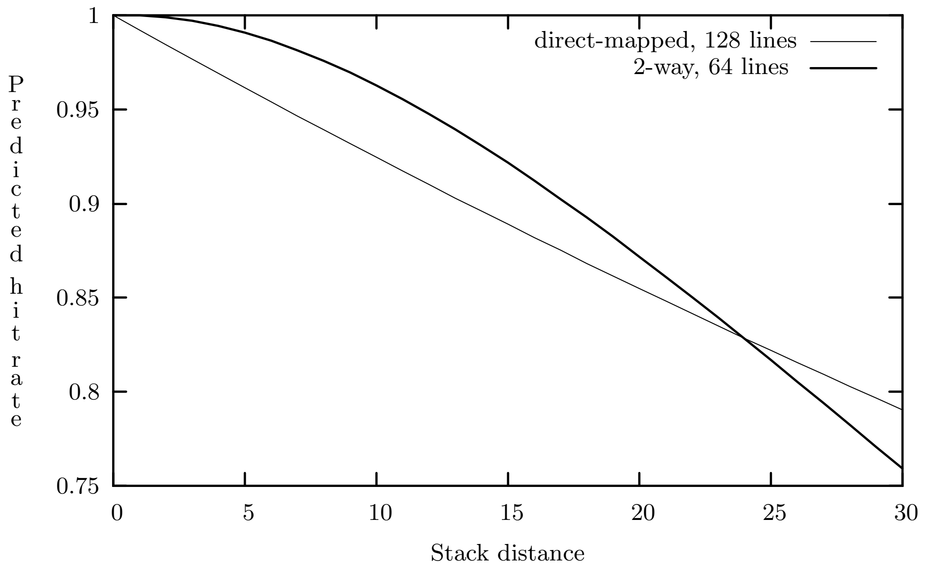


Figure 4.19: Impact of associativity on cache filtering

For example, consider the results from Section 4.4.1 on cache filtering. It was shown that there are very few low-locality accesses in an L2 reference stream. Combining this fact with Figure 4.18 would lead us to believe that this 2:1 Cache Rule is unlikely to apply to L2 caches—the needed locality just is not there.

Measuring conflict in a reference stream

One important issue in caching is the measurement of conflict. The caching models described in this chapter assume that conflict is random, knowing full well that spatial locality will cause the conflict to be less than random while striding and related issues will cause conflict which is greater than random. In this section the issue of quantifying conflict is addressed.

First, the fact that a hit rate can be predicted assuming random conflict allows a trivial measure of the non-random conflict found in the trace. Clearly the difference between the actual hit rate and the predicted hit rate is due to non-random conflict. If the predicted hit rate is greater than the measured hit rate, there is clearly some source of excessive conflict due to the data layout. The problem is that sources of excessive conflict can be masked by sources of scant conflict, so that even if the predicted hit rate exactly matches the actual hit rate there is still quite a bit that could be achieved by changing the data layout. Our assumption is that the sources of excessive conflict involve striding or other heavy use of a single set while scant conflict is due to a high degree of page-level spatial locality.

The actual hit rate reflects the impact of both excessive and scant non-random conflict while the predicted hit rate assumes that neither type of non-random conflict exists. As mentioned in Chapter 2, a “one’s complement” cache (a rather complex hash cache) has the effect of reducing many of the sources of excessive conflict while having very little impact on the scant conflict caused by page-level locality [81]. While it is not exactly true that a one’s complement cache removes all of the bad while leaving the good, it does come fairly close. A new stride length causes excessive conflict (though one that is not a power of two) in a one’s complement cache, and two highly utilized variables could now map to the same set where they didn’t previously. Thus, while a one’s complement cache does not perfectly keep the sources of scant conflict while removing the sources of excessive conflict it does the best of any scheme of which we are aware.

The three hit rates (direct-mapped, one’s complement, and predicted) can be used to provide a rough quantification of conflict. Scant conflict can be measured as the difference between the hit rate of the one’s complement cache (assumed to contain only sources of scant conflict) and the predicted hit rate. Excessive conflict is measured as the difference between the hit rate of the one’s complement cache and the actual hit rate achieved by a direct-mapped cache. Table 4.8 shows the results over the Spec benchmarks. Notice that the excessive conflict is occasionally negative, which is likely due the random chance that two highly used variables conflict which did not conflict in the direct-mapped cache do conflict in the one's complement cache. More rarely the scant conflict is negative. This result is due to the one's complement cache not doing a perfect job of removing excessive conflict.

benchmark	**direct-mapped**	**one's complement**	**predicted**	**excessive conflict**	**scant conflict**
ammp	92.28%	91.80%	90.07%	-0.48%	1.73%
applu	79.27%	79.33%	76.09%	0.05%	3.24%
apsi	75.00%	75.00%	75.01%	0.00%	-0.01%
art	77.04%	77.33%	75.49%	0.30%	1.84%
bzip2	95.20%	95.18%	95.19%	-0.02%	0.00%
crafty	82.17%	83.17%	83.05%	1.00%	0.12%
eon	90.77%	92.59%	90.10%	1.83%	2.49%
equake	94.46%	92.21%	91.99%	-2.26%	0.22%
fma3d	95.40%	95.87%	95.25%	0.47%	0.62%
galgel	98.41%	98.46%	98.28%	0.05%	0.18%
gap	90.83%	90.83%	90.83%	0.00%	0.00%
gcc	89.28%	88.58%	87.22%	-0.70%	1.35%
gzip	95.14%	95.13%	95.13%	-0.02%	0.00%
lucas	97.57%	97.54%	93.29%	-0.03%	4.25%
mcf	90.83%	90.83%	90.83%	0.00%	0.00%
mesa	94.07%	97.21%	93.27%	3.14%	3.94%
mgrid	69.57%	75.44%	75.91%	5.87%	-0.47%
parser	94.10%	94.08%	93.14%	-0.02%	0.94%
perlbmk	85.79%	85.85%	85.65%	0.06%	0.19%
swim	96.81%	96.67%	92.98%	-0.14%	3.68%
twolf	92.79%	95.25%	94.04%	2.47%	1.21%
vortex	92.92%	92.90%	91.99%	-0.01%	0.92%
vpr	98.26%	97.48%	95.55%	-0.78%	1.93%
wupwise	97.31%	97.32%	96.04%	0.00%	1.28%
Average	-	-	-	0.45%	1.24%

Table 4.8: Measurement of excessive conflict

While the above measures are not perfect, we know of no previous work which can determine if a low hit rate is due to poor data layout as opposed to there being a low degree locality[3]. Looking again at the data in Table 4.8, it is obvious that eon, mesa, mgrid and

[3]The 'three C's model' of caching [23, 41], discussed in Chapter 2, does make a rough attempt to distinguish these two concepts but it is so coarse in its granularity as to be nearly worthless as a quantitative measure [13].

twolf have significant room for improvement in their data layout. For a compiler writer this information can provide focus to a study as to what is wrong with the data layout, and could even be used as feedback in a profiling tool. For a cache designer the scant conflict values provide insight into the importance of page-level locality while the excessive conflict values provide insight into the importance of using a hash cache or other technique to remove conflict.

4.4.3 Examining non-standard caches

The last area of application considered is understanding non-standard caches. The ability to *see* how a cache interacts with locality can make comparisons and analysis much easier.

Skew cache replacement policies

In his various papers on skew caches Seznec has proposed a number of replacement schemes including a time-stamp LRU scheme and a single-bit per set [60–62, 73]. In this section those two schemes are compared to a novel scheme which uses partial time-stamps. This novel scheme is evaluated by using the cache modeling techniques presented in this chapter as well as by looking at traditional hit-rate numbers.

Figure 4.20 demonstrates how the skew-cache performance is impacted by changing from the single-bit replacement scheme to a true-LRU scheme under the 100% miss-rate model described in Section 4.3.3 . Clearly the LRU scheme has significantly better performance. The problem with the LRU scheme is that it has fairly high overhead. In a skew cache, unlike a standard cache, any given line might need to be compared to any of the lines in the other way of the cache. To the best of this author's knowledge, no one has formally proposed a method to perform true-LRU replacement in a skew cache. In a conversation with Seznec in

1997, he indicated that the only way his group saw to perform this replacement scheme was using a time stamp. This would be implemented with a counter that is incremented upon every cache access. The line accessed, whether a hit or a miss, would have the current value of that counter stored along with the tag for that data (in the same way LRU information is usually stored). When a replacement decision is required, the 'time stamps' stored with the two potential candidates for eviction would be compared and the smallest value would be the replaced line. When the counter overflows and returns to zero, the cache will either need to be purged or the time stamp values will somehow need to be updated. For a large enough counter this overflow will be very rare (say a few times a minute for a 32-bit counter) and the performance impact small.

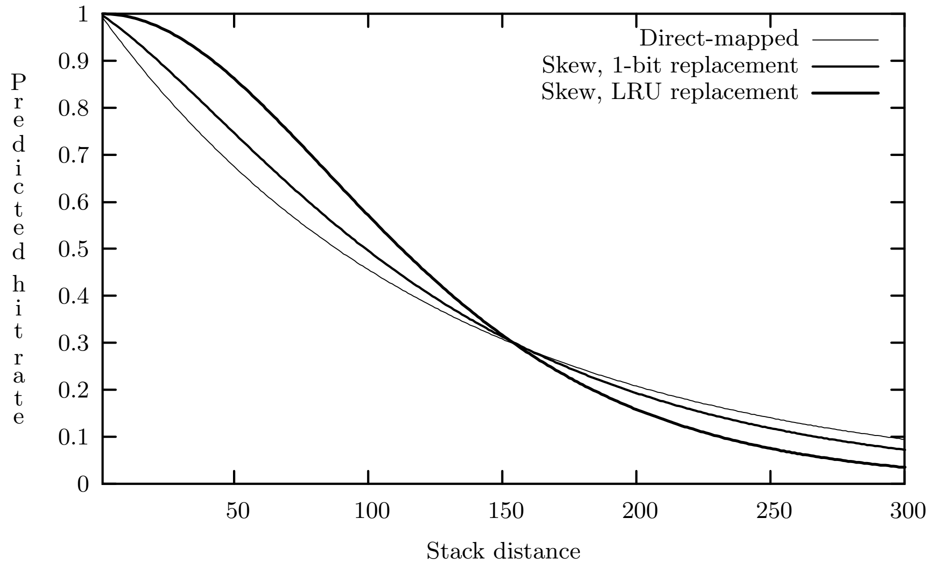


Figure 4.20: Illustration of the impact of the replacement scheme on performance for skew caches. The direct-mapped cache is provided as a reference and all caches consist of 128 lines.

Another option is to use a counter that counts up to a much smaller value but handles the overflow of the counter correctly. In order to get LRU right the vast majority of the time (arbitrarily say 199 times out of 200), the scheme needs to be able to handle about

$4x$ misses where x is equal to the number of lines in the cache[4]. As an example, consider a cache with 128 lines (4 KB) that has an expected hit rate of 90%. Every tenth access will cause a miss. Thus, the counter needs to be able to count up to $10 * 4 * 128$ which requires at least a 13 bit counter.

The 13-bit time stamp would require 208 bytes of additional storage, which increases the amount of non-data storage needed by nearly 50%. (For a 32-bit processor 26 bits would be needed for each tag in addition to state bits consisting of at least a dirty bit and a valid bit.) As the cache gets larger the relative overhead goes up considerably, since as the cache size doubles one more bit is required for LRU information while one less is required for the tag. In any case, the LRU information is well less than 10% of the total size of the cache for any reasonably sized cache.

No matter how large the cache, finding which cache line holds the LRU block requires that the time stamps associated with both cache lines be subtracted from the current counter and then a comparison performed[5]. For the 128 line example this would a 13 bit subtractor followed by a 13-bit comparator. That should likely be fairly similar in speed to the 32-bit comparator proposed above. Including routing to and from the comparator, the replacement decision may take more than one cycle, which should not pose a significant problem as the cache will be waiting for the block to be fetched from the L2 cache (or memory) in any case. The complexity of a multi-cycle replacement policy would be minor but potentially troublesome.

Reducing the number of bits required for the timestamps would have many obvious

[4]The $4x$ value comes from empirical results. It can also be deduced at by graphing the 100% miss rate model for a skew cache and noting that only about 0.5% of the accesses have a stack distance that is greater than $4x$.

[5]While there may appear to be problems with overflow in the subtraction as the time stamp could be smaller than either of the cache line time stamps, in reality this is not a problem as the overflow can be ignored and the correct result will occur when using a standard subtractor.

Replacement Scheme	**Average hit rate**	**Improvement over 1-bit**		
		Average	Min	Max
32-bit time stamp	92.72%	0.21%	-0.14%	1.77%
5-bit time stamp	92.70%	0.19%	-0.31%	0.99%

Table 4.9: A comparison of replacement schemes for skew caches. Hit rate improvement is in terms of absolute improvement over the one-bit replacement scheme.

benefits, the two most significant would be a reduction in the size of the tags and a reduction in the time required to make the replacement decision. The easiest way to perform that reduction would be to store only some of the bits of the time-stamp counter into each of the tags. Experimentation using artificial traces showed that over the Spec benchmarks, storing only bits 6 through 10 of the timestamp (where 0 is the least significant bit) provided performance that was fairly close to that of a 32-bit time stamp. More time-consuming simulation over real traces of length 10 million confirmed this. Table 4.9 show the results over these Spec traces. While the improvement for either scheme is fairly small[6], the much shorter 5-bit time stamps get 90% of the improvement achieved by the full 32-bit time stamps. Table 4.9 show how well this 5-bit scheme compares to the 32-bit timestamp and the 1-bit replacement scheme over all the Spec benchmarks. Clearly the 5-bit scheme does nearly as well on the average as the 32-bit time stamp but suffers somewhat on the extreme cases.

Victim caches

In section 4.3.2 an analytic model for "true-LRU" victim caches was presented. Much like skew caches, time stamps or the equivalently are required to maintain this perfect LRU ordering between the two structures. The other, more standard, scheme is to have the

[6]Seznec saw much more significant differences on the memory reference traces he used. This difference is very much a function of the compiler.

two structures maintain LRU information separately. We call these schemes "true-LRU" and "separate-LRU" respectively. Figure 4.21 shows that the difference between the two schemes is quite large when using the 100% miss rate model. However, the separate scheme performs *much* better when using real data. The reason is that an n-entry fully-associative component of the cache contains the last n evictions. Under the 100% miss rate model these evictions are simply the last n accesses—resulting in the stack distance vs. hit rate graph being the same as the base-cache shifted to the right by n.

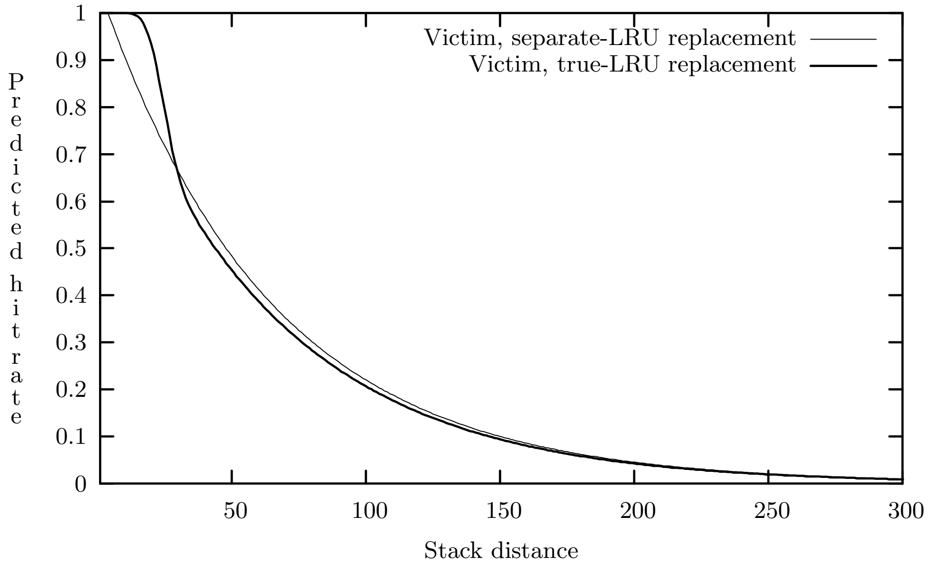


Figure 4.21: Illustration of the impact of the replacement scheme on performance for victim caches.

Under a reference stream with a greater degree of locality, the separate-LRU replacement policy improves significantly while the true-LRU policy sees no improvement at all. Figure 4.22 shows the improvement that occurs when a higher locality reference stream is used, in this case a memory reference stream that has the same locality as gcc. Recall that the true-LRU replacement policy is unaffected by the input locality. That same graph shows the impact of using the same 5-bit time stamp replacement policy discussed in Section 4.4.3

Replacement Scheme	**Average hit rate**	**Improvement over separate-LRU**		
		Average	Min	Max
32-bit time stamp	91.95%	0.11%	-0.17%	0.83%
5-bit time stamp	91.91%	0.08%	-0.41%	0.74%

Table 4.10: A comparison of replacement schemes for victim caches. Hit rate improvement is in terms of absolute improvement.

for the skew cache. As was done in the skew cache section, Table 4.10 shows the impact of the 5-bit time stamp and the 32-bit time stamps when compared to the more standard replacement policy.

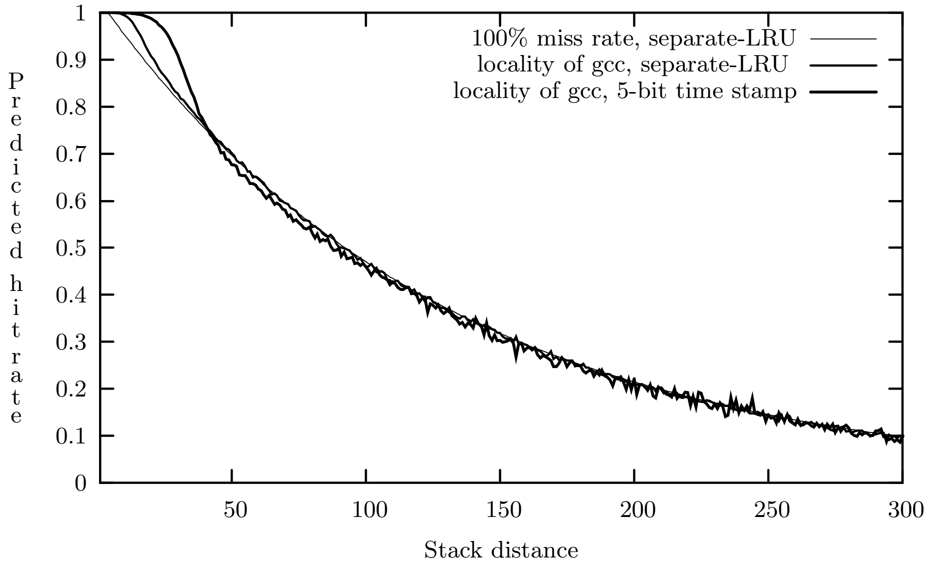


Figure 4.22: Illustration of the impact of locality on the hit rate of the using a 'separate LRU' replacement scheme. This graph also shows the impact of using 5-bit time stamps.

4.5 Future work

There is certainly a large amount of work that can and is being done to expand on the work in this section.

- The ability to identify excessive conflict can be very valuable. It can be used to evaluate the quality of the data layout of a complier. This can help in evaluating compliers in addition to being useful as a profiling technique. Certainly certain reference traces, like our mgrid trace, had excessive conflict. Informing the complier of this excessive conflict so that it can reorder the data layout could be a source of significant performance improvement. Further, comparing the conflict in these reference traces to that found in other work [13] it is obvious that certain compliers simply do a better job laying out data. This can be a helpful piece of information to a complier writer. Preliminary work has started on this, comparing the gcc, MIRV, and lcc compliers. We are currently looking to get the optimized SPEC binaries complied by the Intel Reference Complier.

- Finding an analytic model for multi-level caches has been an insurmountable hurdle. The modeling the interaction of locality and the cache characteristics is extremely complex. While there may be little real value to such a model this work seems somewhat incomplete without it.

- A "better than LRU" replacement policy may be possible using information about the locality of the memory reference stream. While this is almost certainly not realistic to implement in hardware, it may be able to provide something of an upper bound on possible performance under different assumptions.

- Spatial locality is currently only addressed with respect to a single cache line size. Providing a means of understanding spatial locality would be useful when constructing caches that use more than one line size.

- The primary use of the work in this chapter is as a tool to help understand caches and how they interact with locality. The next step in that direction is to make the tools widely available and otherwise be evangelical about the techniques found here. To that end a website is being built as part of an NSF grant. It will provide traces, simulators, graphs, and data to help get the ideas out into the world.

4.6 Conclusion

Every student of computer architecture knows that caches perform well because of spatial and temporal locality. However, few attempts have been made at quantifying this fairly abstract concept of locality. In this chapter a measurement of temporal was proposed with spatial locality being implicitly dealt with though cache line size. This quantification of locality leads to a much fuller understanding than previous models or rules-of-thumb have provided.

In this chapter, an analytic model of how standard caches interact with locality was presented. While very similar models had be proposed in the past by Rau, Smith, Hill and others, we do not know of any attempt to use these models as a method of understanding locality. In addition, novel analytic models of two non-standard caches, LRU skew and true-LRU victim caches, were derived. Next, the 100% miss-rate model of caching was introduced, which makes it possible to find how locality and caching interact using an empirical rather than an analytical model. This 100% miss-rate model allows *any* caching structure to be modeled. That empirical method was also used to help verify that the analytical models were correct. Other than where noted, this work has only been previously seen in our own previous work.

While the primary application of these models is directed toward the understanding of locality and caching, a number of applications were also presented. These applications of the model include being able to measure non-random conflict as well as the ability to generate artificial traces which can be tuned to have whatever locality properties are desired. It is our hope that the novel insight and applications produced in this work prove valuable to those who are involved in cache design.

Chapter 5

Optimal replacement is NP-hard for non-standard caches.

When examining a new cache structure or replacement policy, it is often helpful to use the optimal policy as a baseline. In this chapter it is shown that it is NP-hard to find the optimal schedule for any but the simplest of caching schemes. This is proven by a reduction from 3-Occ-Max-2SAT, a known NP-hard problem [12]. It is also shown that there is no polynomial time algorithm which is guaranteed to find a "good" approximation to this problem unless $P = NP$. These results apply to cache structures that include a victim cache or an assist cache, as well as skew caches and nearly all "multi-lateral" caches [70]. This result also applies to the optimal scheduling of many multi-level cache structures.

5.1 Introduction and Background

One of the greatest challenges of modern computer architecture is the disparity between processor and memory speeds. This disparity has resulted in attempts to eke the last bit of performance out of the entire memory hierarchy, especially caches. As discussed in Chapter 2, victim caches [45], assist caches [17], skew caches [60] and “smart” caches [31, 44, 70, 74, 78] have all been proposed, and some have also been implemented. However, a cache architect’s toolbox has not kept up with these changes, making these caches more difficult to evaluate, understand and improve. In this chapter one of those tools, the optimal replacement algorithm, is examined. It is shown that for many caching structures, it is a tool cache designers and researchers will have to live without.

In 1966 Belady published a heavily cited paper describing the optimal replacement policy for a cache [11]. Other, faster algorithms have been developed which also find the minimal hit rate [49, 67]. As these algorithms perform the same task and generate equivalent results, we refer to this class of optimal replacement algorithms as a single algorithm named MIN. MIN is important because it is used in comparative architectural studies. As was discussed in Chapter 4, MIN has been used in many performance studies [15, 68, 70, 73, 74, 78] and as part of a caching algorithm [78].

MIN is only applicable to “standard caches” such as direct-mapped, set-associative, and fully-associative caches. For many other, more complex, caching schemes the algorithm fails. In those caches it not only matters what block is evicted but where the new block is placed, something MIN does not address. However, until now it has not been clear if a computationally-tractable optimal algorithm could be found for these more complex caches.

The search for a tractable algorithm for optimal placement/replacement in a non-standard cache has existed for a number of years. Most recently, Tam *et al.* [70] published a paper where they could not find a computationally tractable algorithm for optimal placement for their cache design. Instead they made use of a tractable, but sub-optimal, offline algorithm to make the desired comparisons.

5.1.1 Companion caches

Every cache we are aware of has two major components: its *structure* and its *replacement policy.* The structure of a cache describes the physical properties of the cache (such as cache line size, number of cache lines, etc.) as well the legal locations for a given block in the cache. The replacement policy describes when a block is placed into the cache, where that block is placed, and which block, if any, is removed to make room for it. For example, a standard "set-associative cache" uses a set-associative structure and an LRU replacement policy.

The term *Companion Cache Structure* (CCS) is now introduced. A CCS consists of two components: a set-associative or direct-mapped cache and a fully-associative cache. We refer to the direct-mapped/set-associative component as the "main cache" and the fully-associative component as the "companion buffer." Both components must be able to store any arbitrary cache block. A *degenerate* CCS is one in which the main cache is fully-associative or the companion buffer is of size zero. In the first case the CCS reduces to a fully-associative cache. In the second case the CCS is just a standard direct-mapped or set-associative cache.

Many of the non-standard caching schemes discussed in Chapter 2 have a CCS underlying them. Cache structures which include a victim cache or an assist cache with its associated main cache are exactly $CCSs$. Many of the smart caches discussed in Chapter 2 are also CCSs. In addition, skew caches and certain multi-level caches can also be thought of as CCSs. The later two cases will be addressed in Theorems 5.3.7 and 5.3.8 respectively.

In this chapter it is proven that optimally scheduling a non-degenerate CCS is NP-hard with respect to the number of sets in the main cache. We assume that there are no placement restrictions other than those that are part of the CCS. For example, the victim caching scheme, as originally defined [45], consisted of both a structure (a direct-mapped cache and the victim cache itself) as well as a replacement policy.

5.1.2 Related terms

While CCSs encompass a large number of useful cache organizations, it is important to put CCSs in the context of other terms and concepts. Perhaps the most closely related structure to the CCS is the *multi-lateral cache* [70]. A multi-lateral cache consists of multiple cache structures which are conceptually on the same level of the memory hierarchy and which can store the same data. In general, multi-lateral caches contain an underlying CCS, although it is not always obvious.

In a similar way, a *multi-level* cache can also have an underlying CCS. Suppose the L1 cache is fully-associative and the L2 cache is direct-mapped. While these two caches may have very different access times, they can be thought of as a single CCS. Similarly, imagine that rather than being fully-associative, the L1 cache were instead set-associative. As explained in Theorem 5.3.8 this can be thought of as scheduling a number of smaller CCSs. This relies upon having a caching scheme where the L2 is not required to be inclusive of the L1 [9].

Two other terms which are important to the discussion are *bypassing* and internal *reorganization*. Standard caching schemes require that each block be placed in the cache when it is referenced. If this requirement is not in force, the cache is said to allow *bypassing*. Notice that disallowing bypassing places a restriction on the optimal algorithm and cannot improve the miss rate. Another variation is to allow a block to move from location to location without evicting it. We call this *reorganization*. Many modern caching schemes allow for this to some extent; in fact it is at the heart of the replacement schemes for both victim and assist caches as well as certain hash-rehash cache variants [1]. Notice that allowing reorganization cannot increase the miss rate of an optimal algorithm. Further note that reorganization does not affect the miss rate in a standard cache.

5.1.3 Chapter outline

In Section 5.2 it is formally proven that the problem of managing a CCS in order to minimize the number of misses is NP-hard for a very specific instance. That proof is used as a starting point to generalize and expand the results. The specific instance used is a *CCS* where the companion buffer is of size one, the main cache is direct-mapped, bypassing is allowed, and reorganization is not. This result is proven using a reduction from a known NP-hard problem, 3-Occ-Max-2SAT [12]. The expanded results, found in Section 5.3, include:

- The problem is still NP-hard if the companion buffer is of a size greater than one and/or the main cache is set-associative. (Theorems 5.3.1 and 5.3.2)
- The problem is NP-hard whether or not bypassing is allowed. (Theorem 5.3.3)
- Allowing the cache to freely reorganize does not bring the problem into the class P. (Theorem 5.3.4)
- A “good” approximation of OPT for a *CCS* cache is also NP-hard. (Theorems 5.3.5 and 5.3.6)
- Skew and certain multi-level caches are NP-hard to schedule optimally. (Theorems 5.3.7 and 5.3.8)

The net result of all of these proofs is that using optimal scheduling to evaluate replacement algorithms or cache structural changes is not viable for any but the simplest cache

[1]See Chapter 2 for details about these cache types.

structures. This result is important so that cache researchers and designers do not "spin their wheels" looking for a polynomial-time optimal replacement algorithm. If their cache structure has an underlying CCS, they will have to live without that evaluation tool.

5.1.4 Problem definition and goal

Now the problem of offline cache scheduling is described more formally. An ordered list of memory references is sent to the cache. Each reference is to a particular block, and each block can be placed in some subset of the cache. In the offline environment it is known exactly in which order the references will occur. The goal is to minimize the number of misses which occur. A given instance of a caching problem consists of the ordered list of references and a mapping which describes where each reference can be placed in the cache.

5.2 Proofs

In this section the results are proven in three parts, each corresponding to a subsection. In subsection 5.2.1 the relationship between interval scheduling and cache scheduling is described. A formal proof is provided in subsection 5.2.2 for a very restricted problem—when the main cache is direct-mapped, the companion buffer is of size one, no internal reorganization is allowed, and bypassing is allowed. We call this restricted case the Fundamental Companion Cache Scheduling (FCCS) problem. In subsection 5.3 the FCCS proof is used as the foundation for several broader conclusions. These results include showing that the general CCS problem is NP-hard and APX-complete[2]. Also specific bounds are given for how well an algorithm can approximate the FCCS problem.

[2]A problem is APX-complete if it can be approximated within some bound in polynomial time but cannot be approximated to within some arbitrary constant in polynomial time.

The *FCCS* problem is formally defined as follows.

Definition 5.2.1. Fundamental Companion Cache Scheduling(FCCS):

INSTANCE: A direct-mapped cache, D, consisting of sets of locations, $D_1, \ldots D_n$, a companion cache consisting of a single location C_1, a set of memory items (cache lines) M, a mapping $g : M \to D_x$ indicating in which location in the direct-mapped cache each memory item can be stored, a request sequence $s \in M^*$, and an integer K.

QUESTION: Each item of sequence s is selected in order. If that item is currently found in D or C_1, it is counted as a hit. The memory item is then either placed into the location in D indicated by M, placed into C_1, or not placed in the cache at all. Can the memory items be legally placed in the cache such that at least K requests in the sequence s are hits?

5.2.1 Interval scheduling and caching

First, the relationship between cache scheduling and interval scheduling is described. Initially assume that the cache allows bypassing and does not allow internal reorganization (as defined above.) When bypassing is allowed, it is clear that an item need only be put in the cache if it will remain in the cache until its next access. With this in mind, the cache scheduling problem on a cache of size k can be described as a *restricted interval scheduling* problem on k parallel machines.

In short the interval scheduling problem is the following. There is a set of parallel machines and a set of intervals (jobs) that have a fixed starting time and a fixed ending time. A machine can only work on one interval at a time. An interval must be worked on continuously from its start time to its end time to be successfully processed. Each interval can only be processed by a given subset of the machines. The goal is to maximize the number of intervals processed.

The interval scheduling problem relates to caching as follows. The cache locations correspond to machines. Each memory access defines the start time of an interval. The end time of that interval is defined by the next access to that same memory item. This relationship is illustrated in Figure 5.1. Notice that the last access to a given memory item does not start a new interval. Also, notice that each interval corresponds to a *possible* cache hit (that is, the access is not a compulsory miss [39]). The subset of machines that can process an interval is determined by the set of cache locations in which the corresponding memory item can be placed. As stated earlier, when bypassing is allowed, placing an item in the cache is only profitable if the item will still be in the cache the next time it is accessed. Likewise, scheduling an interval on a machine is only profitable if the machine is not otherwise busy during that interval.

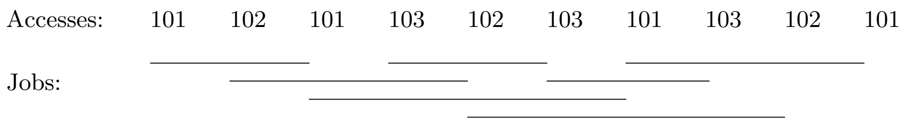


Figure 5.1: Relation between memory accesses and intervals.

If intervals can be placed on any machine, then the interval scheduling is solvable in polynomial time [30]. In the *restricted interval scheduling* problem, each interval can only be placed on a subset of the machines. If the machines can be partitioned into subsets such that if an interval can be scheduled on a machine in one subset, it can be scheduled on all machines in that subset, and it cannot be scheduled on any machine not in that subset, the problem corresponds to a set-associative cache and is still solvable in polynomial time. However, the general case of restricted interval scheduling is NP-complete[7, 48].

In order to show that the FCCS problem is NP-complete, it is first shown that the Fundamental Companion Interval Scheduling problem (FCIS), a special case of restricted interval scheduling, is NP-complete. Then FCIS is reduced to FCCS.

5.2.2 Formal NP-hard proof for FCCS

Definition 5.2.2. Fundamental Companion Interval Scheduling (FCIS)*:*

INSTANCE: A set G of m machines $G_1, \ldots G_m$, a "companion" machine C_1, an integer K, and a set I of n intervals (s_i, f_i, σ_i) where $s_i \in Z^+$ is the starting time of interval i, $f_i \in Z^+$ is the finishing time of interval i and $\sigma_i \in G$ is the machine on which interval i can be scheduled. Additionally, any interval can be scheduled on the companion machine.

QUESTION: Can at least K of the intervals be legally scheduled?

Theorem 5.2.1. *FCIS is NP-complete.*

Proof. 3-Occ-Max-2SAT is reduced to FCIS. 3-Occ-Max-2SAT is a restricted form of Max-2SAT where each variable occurs at most three times.

More formally, 3-Occ-Max-2SAT consists of a set of variables, U, a set of clauses, C, and a value K. Each of the clauses contain exactly two variables, possibly in the negated form. No variable occurs in more than three clauses. As noted above, this problem has been shown to be NP-complete[12].

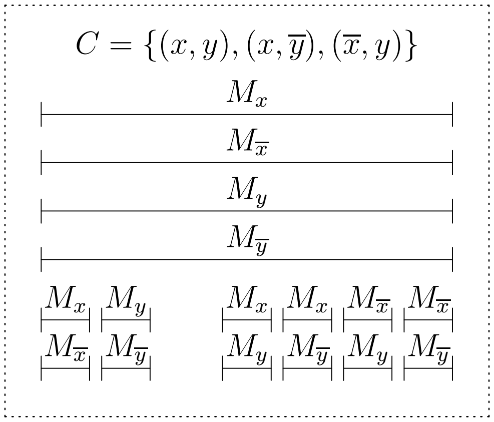
Let $U = \{u_1, u_2, \ldots, u_a\}$, $C = \{c_1, c_2, \ldots, c_b\}$ and K be an instance of 3-Occ-Max-2SAT. For each variable $u \in U$ create two machines, M_u and $M_{\overline{u}}$, both elements of G. For notational reasons, the function $f(x)$ and $h(x)$ are introduced. They are defined as $f(u_i) = i$ and $h(c_k) = k$.

For each variable $u \in U$ create the following four intervals $(f(u), f(u)+1, M_u)$, $(f(u), f(u)+1, M_{\overline{u}})$, $(0, a+b, M_u)$, and $(0, a+b, M_{\overline{u}})$.[3] This results in $4a$ intervals. Label

[3]An alternative notation would be to examine each variable u_i and create intervals $(i, i+1, M_{u_i})$, $(i, i+1, M_{\overline{u_i}})$, $(0, a+b, M_{u_i})$, and $(0, a+b, M_{\overline{u_i}})$

the first two intervals "short variable" intervals and the second two intervals "long variable" intervals. For each clause $c \in C$ create the following two intervals. If variable u appears in clause c in its positive form, create the interval $(a + h(c), a + h(c) + 1, M_u)$. If it appears in the negative form create the interval $(a + h(c), a + h(c) + 1, M_{\overline{u}})$. Since there are two variables per clause, this results in $2b$ "clause" intervals. Thus, a total of $4a + 2b$ intervals are created and $2a$ machines are created. Both are clearly polynomial in the original input size.

An example instance is shown in Figure 5.2. The intervals form an instance of FCIS. The intervals can be broken into three groups: the long variable intervals, the short variable intervals, and the clause intervals. The key observations about this reduction are that the variable intervals (both long and short) are used to enforce a variable assignment and pairs of schedulable clause intervals correspond to the satisfiable clauses.



$C = \{(x, y), (x, \overline{y}), (\overline{x}, y)\}$
M_x
$M_{\overline{x}}$
M_y
$M_{\overline{y}}$
M_x M_y M_x M_x $M_{\overline{x}}$ $M_{\overline{x}}$
$M_{\overline{x}}$ $M_{\overline{y}}$ M_y $M_{\overline{y}}$ M_y $M_{\overline{y}}$

Figure 5.2: Example reduction from 3-Occ-Max-2SAT.
Notice that the short intervals on the left are the "short variable intervals" and the ones on the right are the "clause intervals."

Suppose one can satisfy K of the clauses in the 3-Occ-Max-2SAT instance. It will now be shown that $3a + b + K$ of the intervals can be scheduled. Consider each variable $u \in U$. If u is true, machine $M_{\overline{u}}$ is used to schedule one of the long intervals corresponding to u, and machines M_u and C_1 are used to schedule the two short intervals corresponding to u. If u is false, the roles of machines $M_{\overline{u}}$ and M_u are reversed. Exactly $3a$ of the variable intervals will be scheduled. Note that it is not possible to schedule more than $3a$ of the variable intervals, because for each variable there are 4 overlapping jobs and only 3 available machines.

Now consider each pair of clause intervals. If the corresponding clause c is satisfied in the 3-Occ-Max-2SAT problem, then one of the literals in c must be true. Without loss of generality, it can be assumed that this literal is l. Because l is true, the long interval that could be placed on M_l was not scheduled, and machine M_l is free. Therefore the pair of intervals that correspond to clause c using machines M_l and C_1 can be scheduled. If the clause is not satisfied, only one interval can be scheduled (on C_1). It is important to note that for each u, the clause intervals either use machine M_u or machine $M_{\overline{u}}$ but not both. A total of $b + K$ of the clause intervals will be scheduled. Thus exactly $3a + b + K$ of the intervals have been scheduled.

The proof is only half done at this point. It has been shown that if K of the clauses in our 3-Occ-Max-2SAT problem can be satisfied, one can satisfy $3a + b + K$ of the intervals. It now needs to be shown that if $3a + b + K$ intervals can be scheduled, that schedule can be used to find a solution to the corresponding 3-Occ-Max-2SAT problem where K of the clauses are satisfied.

Suppose there exists a schedule S that schedules $3a + b + K$ intervals. We will create a modified schedule S' that schedules at least $3a + b + K$ intervals with the two following additional constraints: no long variable interval will be scheduled on the companion machine, and for each variable u, exactly one of its long variable intervals will be scheduled.

S' is constructed as follows. First modify S so that C_1 is not used to schedule any long variable intervals. Suppose in schedule S that machine C_1 is used to schedule one of the long intervals for some variable u. In this case, either the corresponding short interval is not currently scheduled or it is scheduled on its corresponding machine in G, M_u or $M_{\overline{u}}$. In the first case, modify S by having C_1 schedule the unscheduled short interval. In the second case, modify S by taking all intervals scheduled on the corresponding machine, M_u or $M_{\overline{u}}$, and schedule them on C_1 and schedule the long interval on that machine. In either case, the modified schedule now completes the same number of intervals as S does, while C_1 is not used to schedule a long variable interval.

Now suppose for a variable u, the modified schedule uses both M_u and $M_{\overline{u}}$ to schedule the corresponding long intervals. This must mean that at least one of the short intervals corresponding to u is not scheduled. Modify the schedule by scheduling one of the unscheduled short intervals on its corresponding machine in G, M_u or $M_{\overline{u}}$, and dropping the corresponding long interval.

At this step there is now a schedule where for each variable u, at most one of its two long intervals is scheduled on its corresponding machine in G, M_u or $M_{\overline{u}}$. The schedule is further modified so that for each variable u, *exactly* one of the two long intervals for u is scheduled on its corresponding machine in G. Let u be a variable where neither of its corresponding intervals are scheduled. Consider the clause intervals. Because the variable u appears at most three times, it either appears in its positive form at most once, or it appears in its negative form at most once. It follows that there is at most one clause interval that can be scheduled on machine M_u or there is at most one clause interval that can be scheduled on machine $M_{\overline{u}}$. Drop that single clause interval and use the freed machine to schedule the corresponding long interval for u. If this machine was currently used to schedule the corresponding short interval, modify the schedule to have C_1 schedule this short interval and the other machine, M_u or $M_{\overline{u}}$, to schedule the other short interval.

Call this final schedule S'. The schedule S' now has at least $3a+b+K$ intervals scheduled since all modifications performed did not decrease the number of intervals scheduled. For each variable, exactly one long variable interval will be scheduled, and it will not be on C_1. This will define a valid truth assignment for 3-Occ-Max-2SAT. Specifically, if the long interval u is not scheduled, then variable u is true; otherwise u is false. Since at most $3a$ variable intervals can be scheduled, this means at least $b+K$ clause intervals are scheduled. At most b of these clause intervals can be scheduled on C_1 which means that at least K clause intervals will be scheduled on a machine in G. These "at least K" clause intervals scheduled on a machine in G correspond to K satisfied clauses in the instance of 3-Occ-Max-2SAT with the above truth assignment. $\square$

Theorem 5.2.2. *The Fundamental Companion Cache Scheduling (FCCS) problem is NP-complete.*

Proof. FCIS is reduced to FCCS. For each machine in FCIS create a cache location in D. For each interval (s_i, f_i, σ_i) create a memory item x_i and let $g(x_i) = \sigma_i$. In other words, memory item x_i can be placed in the cache location that corresponds to the machine on which the interval (s_i, f_i) can be scheduled. Sort the end points $s_1, f_1, s_2, f_2, \dots s_n, f_n$ of the intervals. If two end points are equal, then ties are broken in the following manner. Finishing points come before starting points. Otherwise, the endpoint from the lower indexed interval is first. A sequence of memory references is generated as follows. For each end point in the sorted list, reference the memory element that corresponds to the end point's interval.

The intervals defined by the resulting sequence of memory references and the original set of intervals are equivalent. For every interval that can be scheduled, there will be one cache hit. Therefore, a schedule that schedules more than N intervals will also result in more than

N cache hits. It is trivial to show that the FCCS $\in$ NP, and thus is NP-completesimply by showing that a given proposed solution can be verified to legally schedule K jobs in polynomial time. $\square$

5.3 Extending FCCS

The results of Section 5.2.2 apply only to the FCCS problem—a highly restricted instance of the general problem of optimally scheduling a CCS. This section broadens those results greatly. While it should not be surprising that instances where there are set-associative main caches and larger companion buffers are still NP-hard, the outline of proofs to that effect are provided for sake of completeness. It is also formally proven that allowing internal reorganization, and/or disallowing bypassing, does not affect the difficulty of the problem. Further, it is proven that these problems are all difficult to closely approximate and provide some bounds for the FCCS problem. Lastly, it is proven that skew caches and certain multi-level caches are also NP-hard to schedule optimally.

Theorem 5.3.1. *If the main cache is k-way set-associative, the problem is still NP-complete.*

Proof. The problem statement for FCIS in definition 5.2.2 is used with a slight modification. Each element of G is redefined to be a set of k machines, rather than a single machine. An interval which can be scheduled on G_j may now use any of the (otherwise unscheduled) machines in G_j or the companion machine.

The construction is also modified. Each of the short variable intervals as well as the clause intervals are replicated so that there are k copies of each. Also, in an attempt to reduce confusion, the sets of machines corresponding to the variable x are referred to as S_x

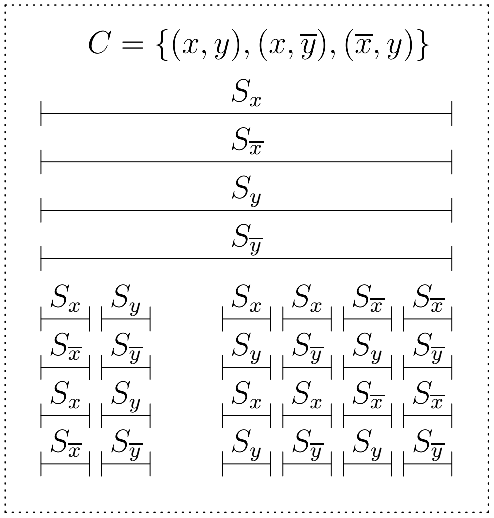


Figure 5.3: Example construction for a 2-way set-associative main cache.

rather than M_x.

Figure 5.3 is an example construction similar to that of Figure 5.2. Notice that exactly one of the variable intervals will still be unschedulable as for each variable there are now $2(k+1)$ variable intervals and only $2k+1$ machines on which they can be run. Also notice that if the long variable jobs corresponding to the literals of a given clause are both scheduled, one of the clause jobs corresponding to the clause will be unschedulable. Thus the same basic arguments of Theorem 5.2.1 still apply, and the conversion from an interval scheduling problem to a set-associative caching problem is only a simple variation of Theorem 5.2.2.

□

Theorem 5.3.2. *If the companion buffer is of size $z > 1$, the problem is still NP-complete.*

Proof. The problem statement for FCIS in definition 5.2.2 is used with a slight modification. The single companion machine C_1, is now a set C of companion machines $C_1 \ldots C_z$. Much like before, any job can be run on any of the machines in C.

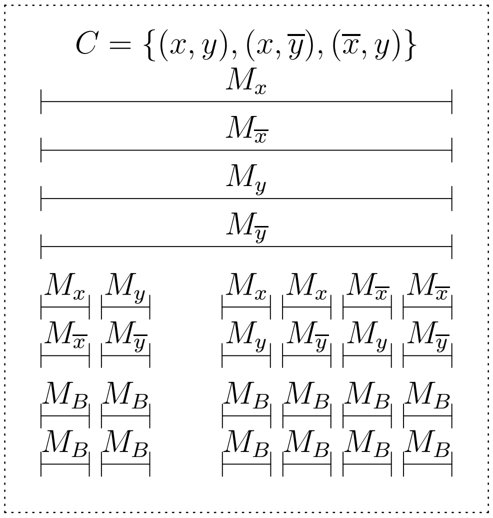

$C = \{(x,y),(x,\overline{y}),(\overline{x},y)\}$
M_x
$M_{\overline{x}}$
M_y
$M_{\overline{y}}$
M_x M_y M_x M_x $M_{\overline{x}}$ $M_{\overline{x}}$
$M_{\overline{x}}$ $M_{\overline{y}}$ M_y $M_{\overline{y}}$ M_y $M_{\overline{y}}$
M_B M_B M_B M_B M_B M_B
M_B M_B M_B M_B M_B M_B

Figure 5.4: Example construction for a companion buffer of size 3

The construction used in Theorem 5.2.1 is used with slight modifications. A new machine, $M_B \in G$ is added. For each pair of short intervals, add $z-1$ intervals with the same start and end times as that pair of intervals and which can run on machine M_B. Figure 5.3 illustrates the modified construction for a companion buffer of size 3.

If all of these "B" intervals are scheduled they will use the $z-1$ extra machines of the companion buffer. When creating the schedule S' as described in Theorem 5.2.1 an additional step is added. After the long variable intervals have been fixed, any unscheduled B intervals will be scheduled if a free machine is available. If no such machine is available, it must be the case that both of the non-B short interval jobs are using a machine from the set C. Unschedule one of those intervals and let the unscheduled "B" interval use that machine. The remainder of the construction is identical. ☐

Theorem 5.3.3. *If bypassing is not allowed in the cache, the problem is still NP-complete.*

Proof. This proof uses nearly the same construction as Theorem 5.2.1. The only change is that the long intervals are lengthened so they start at time -1 rather than time 0. Notice that this change does not have any impact on either of the theorems used to show that $FCCS$ is NP-hard. All that is left is to prove that disallowing bypassing will not decrease the number of intervals which can be scheduled by an optimal algorithm. Clearly, disallowing bypassing cannot increase the optimal number of intervals which can be scheduled in any instance: any schedule legal under the no-bypassing case is also a legal schedule with bypassing.

Let us now define the following terms. I is an instance of the FCIS problem. $S^{OPT}_{bypass}(I)$ is the optimal schedule for I when bypassing is allowed. Further we define $S^{OPT}_{bypass}(I)$ to be of the form of S' described in Theorem 5.2.1. That is, all of the short variable jobs are scheduled, one long variable job is left unscheduled for each variable, and some of the clause intervals may be left unscheduled. That theorem showed that such a schedule exists which is still optimal.

Notice that under the no-bypassing restriction, it must be the case that all of the short intervals are scheduled. That is because the short intervals must be started and no other interval starts while they run, so no interval can preempt them. Clearly the short intervals may be preempting the long intervals. In fact the only impact of disallowing bypassing for any I is that all of the short intervals *must* be scheduled: as long as that rule is followed the schedule will be a legal non-bypassing schedule. The long intervals must all initially be scheduled, but may be preempted later, and thus count as unscheduled. This means that if S^{OPT}_{bypass} can be converted to a form where the long jobs are the only jobs left unscheduled and the total number of jobs scheduled does not decrease then there exists a schedule that does not require bypassing and still schedules exactly as many jobs.

A schedule $S^{OPT}_{nobypass}$ is now constructed from the schedule S^{OPT}_{bypass}. Label the k unscheduled clause intervals in S^{OPT}_{bypass} as $UCI_1 \ldots UCI_k$ and label the long variable interval which uses the same machine in G used by UCI_j as LVI_j. UCI_j could be scheduled if LVI_j were unscheduled. That is because LVI_j is the only interval which can possibly use its machine in G at that time. Since LVI_i overlaps with at least all of the intervals that UCI_j overlaps with, it cannot decrease the number of schedulable jobs to schedule UCI_j and unschedule LVI_j. This can be done for each of the k unscheduled clauses and will result in no fewer intervals being scheduled.

It has been shown that $S^{OPT}_{nobypass}$ schedules exactly as many intervals as S^{OPT}_{bypass}. As finding S^{OPT}_{bypass} is NP-hard, finding $S^{OPT}_{nobypass}$ must be NP-hard also. $\square$

Theorem 5.3.4. *If reorganization is freely allowed in the cache, the problem is still NP-complete.*

Proof. The same construction as Theorem 5.2.1 is used once again. Notice that this problem is *less* restricted than the FCIS problem. Thus the optimal schedule for the reorganization variation can be no worse than the optimal non-reorganization schedule for the same instance. It will be shown that allowing reorganization does not improve the number of intervals which can be scheduled and therefore FCIS where reorganization is allowed must also be NP-hard.

The fundamental observation is that given m intervals, all of which desire to be run during some given time, and $m - 1$ machines on which to run those intervals, one of the intervals must be left unscheduled. Reordering does not help in that case. Another important observation is that once an interval is left unscheduled, even for a moment, there is no benefit to scheduling it in the future. Using these facts, it will be shown that the proof in Theorem 5.2.1 holds for the reorganization case also. First it is shown that no extra variable intervals can be scheduled due to reorganization. Next it is shown that no extra clause intervals can be scheduled due to reorganization. As that is all of the intervals in the instance, it is enough to show that reorganization never allows extra intervals can be scheduled in the optimal cache for any FCIS instance.

As before, at least one of the variable intervals must be left unscheduled. Again, leaving a short variable interval unscheduled can be no better than leaving a long variable interval unscheduled, and thus it can be assumed that one long variable interval is left unscheduled per variable. It is therefore clearly the case that just before the first clause interval has been scheduled exactly as many intervals can be scheduled in the standard FCIS version as in the FCIS version where reorganization is allowed.

It is now shown that no extra clause intervals can be scheduled due to reorganization. Label each of the pairs of short clause intervals as (CI_{l1}, CI_{l2}) where CI_{l1} can be assigned to machine $l1 \in G$ and CI_{l2} is defined similarly. Further, the term LVI_x will be used to mean the long variable interval which can be run on machine $x \in G$. Examine each pair of clause intervals in turn. As before, if LVI_{l1} and/or LVI_{l2} was left unscheduled, both CI_{l1} and CI_{l2} can be scheduled. If LVI_{l1} and LVI_{l2} are both scheduled, there exist four simultaneous intervals (CI_{l1}, CI_{l2}, LVI_{l1} and LVI_{l2} which can only be run on three machines (M_{l1}, M_{l2}, and C_1). Thus, as in Theorem 5.2.1, one of those four intervals cannot be scheduled.

The only issue which remains is to show that there is no advantage to leaving two long variable intervals associated with the same interval unscheduled. The argument for this is identical to the argument presented in Theorem 5.2.1: if both long variable intervals are left unscheduled, the one for which there exists only one clause interval which uses its machine in G can be scheduled while that clause interval is unscheduled. Recall that this is due to the 3-occurrence restriction of 3-Occ-Max-2-SAT. Therefor, the same number of intervals are scheduled by the optimal solution to FCIS with reorganization as are scheduled by standard FCIS. As FCIS is NP-complete, FCIS with reorganization must also be NP-complete. $\square$

Theorem 5.3.5. *Any algorithm which can guarantee to approximate the miss rate within 0.0124% of the optimal for FCIS must have an exponential running time if $P \neq NP$.*

Proof. Berman and Karpinski proved that it is NP-hard to approximate 3-Occ-Max-2SAT within 0.05% [12]. Specifically, they showed that there exist instances of $2016n$ clauses where it is NP-hard to compute if the number of satisfiable clauses is $(2011 + \epsilon)n$ or $(2012 + \epsilon)n$ for any ϵ in the range $(0, 0.5)$. Utilizing the construction in Theorem 5.2.1 it is possible to take an instance of these $2016n$ clauses and convert it into a FCIS problem. Recall from the construction in Theorem 5.2.1 that $4a + 2b$ clauses were generated when the 3-Occ-Max-2SAT problem consisted of a variables and b clauses.

Because the instances generated in [12] all utilize each variable exactly three times, there are exactly $1344n$ distinct variables used. Therefore, there will be exactly $(4 * 1344 + 2 * 2016)n = 9408n$ clauses in the FCIS instance. Again, recall from Theorem 5.2.1 that if K clauses in the 3-Occ-Max-2SAT problem could be satisfied, $3a + b + K$ clauses in the FCIS instance could be scheduled. For $K = (2012 + \epsilon)n$ and $(K = 2011 + \epsilon)$, that means that $(8060 + \epsilon)n$ and $(8059 + \epsilon)n$ could be scheduled respectively. Clearly, as n gets large and ϵ can therefore approach 0, it is NP-hard to distinguish between $8060n$ and $8059n$ schedulable intervals. It is then the case that estimation of FCIS within a factor of 8060/8059 is NP-hard. This is the same as being unable to estimate within 0.0124% of the optimal.

It should be noted that the technique used in Theorem 5.3.6 could easily be used to compute a similar result. However that generic technique yields a looser result of 0.0079%.

Theorem 5.3.6. *An algorithm which can guarantee to approximate the miss rate within 1.01% of the optimal for FCCS is non-trivial to find.*

According to Berman and Karpinski [12], the best known polynomial-time approximation algorithm for 3-Occ-Max-2SAT is the same as that for Max-2-SAT found by Feige and Goemans [27]: 1.0741. Clearly, as FCIS can be used to solve 3-Occ-Max-2SAT, this result could be improved if there were a sufficiently good approximation algorithm for FCIS. It is now shown that if there exists a 1.013-algorithm for FCIS (that is, it can approximate FCIS within 1.01%) it would result in an improvement in the best known approximation algorithm for 3-Occ-Max-2SAT. As this problem has been fairly well studied, we are comfortable labeling such an improvement in 3-Occ-Max-2SAT as non-trivial.

An L-reduction, as explained by Papadimitriou in [55], can be used to show how an inapproximability result for one problem can be reduced to another. To quote from various parts of pages 309–311[4].

> Suppose that A and B are optimization problems (maximization or minimization). An *L-reduction* from A to B is a pair of functions R and S, both computable in logarithmic space, with the following two additional properties: First, if x is an instance of A with optimum cost $\mathrm{OPT}(x)$, then $R(x)$ is an instance of B with optimum cost that satisfies
>
> $$\mathrm{OPT}(R(x)) \leq \alpha \cdot \mathrm{OPT}(x) \tag{5.1}$$
>
> where α is a positive constant. Second, if s is any feasible solution of $R(x)$, then $S(s)$ is a feasible solution of x such that
>
> $$|\mathrm{OPT}(x) - c(S(s))| \leq \beta |\mathrm{OPT}(R(x)) - c(s)|, \tag{5.2}$$

[4]We have replaced Papadimitriou's (non-standard) use of ϵ with the symbol δ. Otherwise the quotes are verbatim.

where β is another positive constant particular to the reduction (and we use c to denote that cost in both instances).

Papadimitriou goes on to state:

> If there is an L-reduction (R, S) from A to B with constants α and β, and there is a polynomial-time δ-approximation algorithm for B, then there is a polynomial-time
> $$\frac{\alpha\beta\delta}{1-\delta} \tag{5.3}$$
> -approximation algorithm for A.

It should be noted that Papadimitriou uses the term x-approximation algorithm in a non-standard way. He defines an x-approximation algorithm as:

$$x = \frac{|c(M(x)) - \mathrm{OPT}(x)|}{\mathrm{OPT}(x)},$$

rather than the more standard form of $x = c(M(x))/\mathrm{OPT}(x)$. For the remainder of this section the term "x-approximation algorithm," where x is a real number, will refer to the standard definition. The symbols δ and ϵ will be used to refer to the inapproximablilty using Papadimitriou's or the standard scheme respectively. Simple algebra can show that

$$\delta = \frac{\epsilon - 1}{\epsilon} \tag{5.4}$$

$$\epsilon = \frac{1}{1-\delta} \tag{5.5}$$

Thus, if A is 3-Occ-Max-2SAT and B is FCIS, there is an ϵ-approximation for A of 1.0741. That ϵ-approximation is the same as a δ-approximation of about 0.069. It is now shown that if an ϵ-approximation algorithm of less than 1.013 exists for B, an ϵ of less than 1.0741 also exists for A. This would be an improvement in the best-known algorithm for 3-Occ-Max-2SAT.

The value of β from Equation 5.2 is now found via a two-part argument. Let s be a feasible solution of FCIS which schedules i intervals and let m be the optimal number of intervals that could have been scheduled in that instance. First assume that s schedules at least $3a + b$ intervals, where the instance consists of a variables and b clauses. Recall from Theorem 5.2.1 that if $3a+b+K$ clauses in FCIS could be satisfied, at least K clauses in the 3-Occ-Max-2SAT instance could be scheduled. Thus, $\mathrm{OPT}(x) - c(S(s))| = m - c$ and $|\mathrm{OPT}(R(x)) - c(s)| = (3a + b + m) - (3a + b + c) = m - c$ and from Equation 5.2, $\beta = 1$. Now assume that instead s schedules fewer than $3a + b$ intervals. In that case, even if no 3-Occ-Max-2SAT clauses are satisfied, β is clearly less than or equal to one. That is because $\mathrm{OPT}(x) - c(S(s))| < m \leq |\mathrm{OPT}(R(x)) - c(s)|$.

Computing α requires one additional fact: At least $3/4$ of all clauses in a Max-2SAT problem can always be scheduled. Given that, if m is the optimal solution for the 3-Occ-Max-2SAT problem, there could have been no more than $4m/3$ clauses in the problem instance. Further if a variable only occurs once, that literal can safely be set to true and the problem reduced in size by one variable and one clause. It can therefore be assumed that each variable occurs at least twice. If, as before, a is the number of variables in the instance and b is the number of clauses, $a \leq 4m/3$ and $b \leq 4m/3$. Thus, the FCIS problem can schedule exactly $3a + b + m = 12m/3 + 4m/3 + m = 19m/3$ intervals. Plugging the appropriate values into Equation 5.1, $\alpha = 19/3$.

Having α and β available, and using Equations 5.3,5.5, and 5.4 it can be shown that, if a better than 1.0101-approximation algorithm for FCIS were found, that would lead to an improvement in the best known approximation algorithm for 3-Occ-Max-2SAT[5]. This does not mean that there is no polynomial time algorithm which can guarantee a result of better than 1.6%. However, such a result would show an improvement over the best known algorithm for a fairly well-studied problem. We are comfortable labeling such an approximation algorithm "non-trivial," □

Theorem 5.3.7. *Finding the optimal schedule for a skew cache is NP-hard.*

Proof. This is proven by restriction [28]. Notice that it is possible to have a schedule where all of the accesses to one of the banks of the skew cache are always to the same location. In that restricted case the problem is exactly the same as FCCS. Thus if an algorithm existed which could optimally schedule a skew cache, that same algorithm could be used to schedule a FCCS. Since FCCS is NP-hard, skew cache scheduling is also NP-hard. □

Theorem 5.3.8. *It is NP-hard to optimally schedule a* multi-level *cache as a single unit where exclusion is allowed.*

Proof. This is also a proof by restriction [28]. Assume a two-level cache hierarchy where the L1 cache can hold k_1 blocks and the blocks are grouped into s_1 sets. Define s_2 and k_2 in a similar way for the L2 cache. Without loss of generality we will assume that $s_2 > s_1$[6]. Table 5.3 lists the values of s_2/s_1, for a few selected processors. Values near 8 or 16 are most common.

Consider the set of blocks, B_i, which are mapped into the ith set of the L1 cache. Under all reasonable indexing functions (see Chapter 2 for a discussion on indexing functions) elements of B_i will also be mapped into at least s_2/s_1 different sets of the L2 cache. Now

[5]This solution can be easily verified. However *finding* the value 1.0101 required some additional work
[6]With only slight modifications The same argument can be used if $s_1 > s_2$

Table 5.1: Ratio of the number of sets in the L2 cache over the number in the L1 data cache for selected processors.

Processor	s_2/s_1
Intel P6 w 512KB L2 cache	64
Intel Celeron (Coppermine)	8
Compaq 21364	8
AMD 1GHz Athlon	0.5

notice that optimally scheduling accesses to the blocks in the set B_i is exactly the same as scheduling a companion cache, where the companion buffer is of size k_1/s_1, and the main cache consists of at least s_2/s_1 sets each of size k_2/s_1.

It is worth noting that for small values of s_2/s_1 the problem may be computational tractable. In fact, for a fixed value of s_2/s_1 the problem is technically requires only polynomial run time. However, the large values for s_2/s_1 in real processors, combined with the very large address traces (usually greater than 1,000,000 addresses) would seem to make this problem generally intractable. $\square$

5.4 Conclusion

It has been shown that an important tool for cache studies, the optimal replacement policy, is NP-hard to compute for many non-standard caches. It has also been shown that this result applies to the optimal scheduling of multi-level caches where the caches are kept exclusive. Additionally, it was proven that a good *approximation* to these problems is also NP-hard to compute. While these results will not help one build faster caches, researchers who are aware of this will not waste their time in a futile attempt to find efficient algorithms for these problems. Unless $P = NP$ such algorithms simply do not exist.

Bibliography

Bibliography

[1] A. Agarwal, J. Hennessy, and Horowitz M. Cache performace of operating systems and multiprogramming. *ACM Trans. on Computer Systems*, 6(4):393–431, November 1988.

[2] A. Agarwal, J. Hennessy, and Horowitz M. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), May 1989.

[3] A. Agarwal and S. D. Pudar. Column-associative caches: a technique for reducing the miss rate of direct-mapped caches. In *Proc. of the 20th Int'l Symposium on Computer Architecure*, pages 179–190, 1993.

[4] V. Agarwal, M.S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate verse IPC:The end of the road for conventional microarchitecures. In *Proc. of the 27th Int'l Symposium on Computer Architecture*, June 2000.

[5] E. Anderson, P.V. Vleet, L. Brown, J-L Baer, and A.R. Darlin. On the performace potential of dynamic cache line sizes. Technical Report UW-CSE-99-02-01, Dept. of Computer Science and Engineering, University of Washington, February 1999.

[6] J. Archibald and J-L. Baer. Cache coherence proto-cols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, November 1986.

[7] E.A. Arkin and E.B Silverberg. Scheduling jobs with fixed start and end times. *Discrete Applied Mathematics*, 18:1–8, 1987.

[8] O. Babaoglu. Efficient generation of memory reference strings based on the lru stack model of program behaviour. In *PERFORMANCE '81*, pages 373–383, 1981.

[9] J-L. Baer and W-H. Wang. On the inclusion properties of multi-level cache hiearchies. In *Proc. of the 15th Int'l Symposium on Computer Architecture*, June 1988.

[10] A.P. Batson and A. W. Madison. Characteristics of program locality. *Communications of the ACM*, 19(5):285–294, May 1976.

[11] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):282–288, 1966.

[12] P. Berman and M. Karpinski. On some tighter inapproximability results. Technical Report 99-23, DIMACS, 1999.

[13] M. Brehob and R.J. Enbody. An analytical model of locality and caching. Technical Report MSUCPS:TR99-31, Michigan State University, Department of Computer Science and Engineering, 1999.

[14] M. Brehob, R.J Enbody, and N. Wade. Analysis and replacement for skew-associative caches. Technical Report MSUCPS:TR97-32, Michigan State University, Department of Computer Science and Engineering, 1997.

[15] D. Burger, J.R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *23nd Annual Int'l Symposium on Computer Architecture*, pages 78–89, 1996.

[16] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proc. 2nd Int'l Symposium on High Performace Computer Architecture*, pages 244–253, 1993.

[17] K.K. Chan, C.C. Hay, J.R. Keller, G.P. Kurpanek, F.X. Schumacher, and J. Zheng. Design of the HP PA7200. *Hewlett-Packard Journal*, February 1996.

[18] J.H. Chang, H. Chao, and K. So. Cache design of a sub-micron CMOS system/370. In *Proc. of the 14th Int'l Symposium on Computer Architecture*, pages 179–190, 1987.

[19] C. K. Chow. An optimization of storage hierarchies. *IBM Journal of Research and Development*, 18(3):194–203, May 1974.

[20] C.K. Chow. Determination of cache's capacity and its matching storage hierarchy. *IEEE Transactions on Computers*, C-25(2):157–164, February 1976.

[21] Standard Performace Evaluation Corporation. Spec benchmarks. World Wide Web.

[22] D.E. Culler. Caches. World Wide Web, February 2002.

[23] Gee J. D., Hill M. D., Pnevmatikatos D. N., and Smith A. J. Cache performance of the SPEC92 benchmark suite. *IEEE Micro*, 13(4):17–27, August 1993.

[24] P.J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5), May 1968.

[25] Wilton S. J. E. and Jouppi N. P. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid-State Circuits*, 31(5):677–688, May 1996.

[26] and P.J. Denning E.G. Coffman. *Operating Systems Theory.* Prentice-Hall, 1973.

[27] Uriel Feige and Michel Goemans. Approximating the value of two prover proof systems, with applications to max 2sat and max dicut. In *In Proc. of 3rd Israel Symposium on the Theory of Computing and Systems*, pages 182–189, 1995.

[28] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, 1973.

[29] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proc. of the 11th ACM Int'l Conference on Supercomputing*, July 1997.

[30] M.C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[31] A. Gonzalez, C. Aliagas, and M. Valero. Data cache with multiple caching strategies tuned to different types of locality. In *Proc. of the 1995 Conference on Supercomputing*, pages 338–347, 1995.

[32] K. Grimsrud. *Quantifying Locality*. Doctoral dissertation, Brighman Young University, 1993.

[33] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. On the accuracy of memory reference models. *Lecture Notes in Computer Science*, 794:369–388, 1994.

[34] K. Grimsrud, J. Archibald, R. Frost, and B. Nelson. Locality as a visualization tool. In *IEEE Transactions on Computers*, volume 45, pages 1319–1325, November 1996.

[35] M. Hachman. IBM and Intel chip vendors showcase road maps at microprocessor forum. *BYTE*, October 1999.

[36] E. Hallnor and S. K. Reinhardt. A fully associative software-managed cache design. In *27th Int'l Symposium on Computer Architecture*, June 2000.

[37] J. Handy. *The Cache Memory Book*. Academic Press, second edition, 1998.

[38] J.P. Hayes. *Computer Architecture and Organization*. McGraw Hill Text, 1988.

[39] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, third edition, 2002.

[40] M.D. Hill. A case for direct-mapped caches. *IEEE Computer*, pages 25–40, December 1988.

[41] M.D. Hill. Evaluating associativity in cpu caches. *IEEE Trans. on Computers*, 38(12):1612–1630, December 1989.

[42] Micron Technology Inc. *1997 DRAM Data Book*. Micron Technology Inc., Boise, Idaho, 1997.

[43] B. Jacob, P. Chen, S. Silverman, and T. Mudge. An analytical model for designing memory hierarchies. *IEEE Transactions on Computer*, 45(10), Oct 1996.

[44] T.L. Johnson and W-M. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proc. of the 24th Annual Int'l Symposium on Computer Architecture*, volume 25,2 of *Computer Architecture News*, pages 315–326, New York, June 1997. ACM Press.

[45] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *Proc. of the Seventeenth Annual International Symposium on Computer Architecture*, 18(2):364–373, May 1990.

[46] S.F. Kaplan, Y. Smaragdakis, and P. R. Wilson. Trace reduction for virtual memory simulations. In *Proc. of ACM SIGMETRICS*, May 1999.

[47] R. R. Kessler and et al. Inexpensive implementations of set-associativity. In *Proc. of the 16th Int'l Symposium on Computer Architecture*, pages 131–139, 1989.

[48] A.W.J. Kolen and J.G. Kroon. On the computation complexity of (maximum) class scheduling. *European Journal of Operational Research*, 54:23–38, 1991.

[49] R.L. Mattson, J. Gecsei, D. R. Slutz, and I.L. Traiger. Evaluation techiques for storage hierarchies. *IBM system Journal*, 9(2), 1970. Periodical Room, TA168 .I14 v.12 1973.

[50] S.A. McKee, A. Aluwihare, B.H. Clark, R.H. Klenke, T.C. Landon, C.W. Oliver, M.H. Salinas, A.E. Szymkowiak, K.L. Wright, W.A. Wulf, and J.H. Aylor. Design and evaluation of dynamic access ordering hardware. In *Proc. 10th ACM Int'l Conf. on Supercomputing*, May 1996.

[51] S.A McKee, R.H. Klenke, K.L. Wright, W.A. Wulf, M.H. Salinas, J.H. Aylor, and A.P. Batson. Smarter memory: Improving bandwidth for streamed references. *IEEE Computer*, July 1998.

[52] S.A. McKee, W.A. Wulf, J.H. Aylor, R.H. Klenke, S.I. Salinas, M.H. Hong, and Weikle D.A.B. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, To appear.

[53] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Cambridge, Massachusetts, October 1996. ACM Press.

[54] G. D. McNiven and E. S. Davidson. Analysis of memory referencing behavior for design of local memories. In H. J. Siegel, editor, *Proc. of the 15th Int'l Symposium on Computer Architecture*, pages 56–63, Honolulu, Hawaii, May 1988. IEEE Computer Society Press.

[55] C.H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 229–234, 1988.

[56] D. A. Patterson, J. L. Hennessy, and D. A. Peterson. *Computer Organization and Design : The Hardware/Software Interface.* Morgan Kaufmann, San Mateo, CA, second edition, 1997.

[57] S.A. Przybylski. *Cache memory hierarchy design: A Performance-Dircted Approoch.* Morgan Kaufmann, 1990.

[58] R. Rau. Properties and applications of the least-recently-used stack model. Technical Report 139, Dept. of Electical Engineering, Stanford Univ., May 1977.

[59] D.L. Rhodes and W.Wolf. Unbalanced cache systems. In *Proc. of the IEEE Int'l Workshop on Memory Technology, Design and Testing*, pages 16–23, 1999.

[60] A. Seznec. A case for two-way skewed-associative caches. In *Proc. of the 20th International Symposium on Computer Architecture*, pages 169–178, 1993.

[61] A. Seznec and F. Bodin. Skewed-associative caches. In *Proc. of PARLE 93*, 1993.

[62] A. Seznec and F. Bodin. Skewed associativity enhances performance predictablility. In *Proc. of the 22nd International Symposium on Computer Architecture*, pages 256–275, 1995.

[63] A Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts.* Addison-Wesley, third edition, 1991.

[64] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[65] A.J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Transactions on Software Engineering*, 4(2):121–130, March 1978.

[66] H.S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, 1990.

[67] R.A. Sugumar. *Multi Configuration Simulation Algorithms for the Evaluation of Computer Architecture Designs.* Doctoral dissertation, University of Michigan, 1993.

[68] R.A. Sugumar and S.G. Abraham. Efficient simulation of caches under optimal replacement with appliations to miss characterization. In *Proc. of ACM SIGMETRICS*, May 1993.

[69] E.S. Tam, J.A. Rivers, V. Srinivasan, G.S. Tyson, and E.S. Davidson. Evaluating the performance of active cache management schemes. In *Proc. of the 1998 IEEE International Conference on Computer Design*, pages 368–375, October 1998.

[70] E.S. Tam, J.A. Rivers, V. Srinivasan, G.S. Tyson, and E.S. Davidson. Active management of data caches by exploiting reuse information. *IEEE Trans. on Computers*, 48(11), November 1999.

[71] D. Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 38(7), July 1989.

[72] E. Torng. A unified analysis of paging and caching. *Algorithmica*, 20:175–200, 1998.

[73] D. Truong, F. Bodin, and A. Seznec. Accurate data layout into blocks may boost cache performance. In *Interact-2*, February 1997.

[74] G. Tyson, M. Farrens, J. Matthews, and A.R. Pleszkun. A modified approach to data cache management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, Ann Arbor, Michigan, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[75] R.A. Uhlig and T.N. Mudge. Trace-driving memory simulation: a survey. *ACM Computing Surveys*, 29(2):128–170, June 1997.

[76] D.A.B Weikle, S.A. McKee, and W.A. W.A. Wulf. Caches as filters: A new approach to cache analysis. In *Proc. of the Sixth International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, July 1998.

[77] Dee A. B. Weikle, Kevin Skadron, Sally A. McKee, and W. A Wulf. Caches as filters: A unifying model for memory hierarchy analysis. Technical Report CS-2000-16, Dept. of Computer Science, University of Virginia, June 2000.

[78] W.A. Wong and J-L. Baer. Modified lru policies for improving second-level cache behavoir. In *6th Int'l Symposium on High-Performance Computer Architecture*, January 2000.

[79] C.E. Wu, Y. Hsu, and Y-H Liu. A quantitative evaluation of cache types for high-performace computer systems. *IEEE Trans. on Computers*, 42(10), October 1993.

[80] W.A. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23:20–24, 1995.

[81] Q. Yang and S. Adina. A one's complement cache memory. In *Proceedings of the 23rd International Conference on Parallel Processing. Volume 1: Architecture*, pages 250–257, Boca Raton, FL, USA, August 1994. CRC Press.

[82] C. Zhang, X. Zhang, and Y. Yan. Two fast and high-associativity cache schemes. *IEEE Micro*, 17(5):40–49, September 1997.