MISS REDUCTION IN LARGE, REAL-INDEXED CACHES

by

R.E. Kessler and Mark D. Hill

Computer Sciences Technical Report #940

June 1990

Miss Reduction in Large, Real-Indexed Caches[†]

R. E. Kessler
Mark D. Hill

University of Wisconsin
Computer Sciences Department
Madison, Wisconsin 53706
kessler@cs.wisc.edu
markhill@cs.wisc.edu

ABSTRACT

In computer systems using paged virtual memories, the virtual to real page mapping can have an effect on the performance of large, real-indexed caches. We will show the effect on cache miss frequency is significant and examine implementable schemes to produce page mappings which will eliminate many misses in these caches. A simple model is developed to indicate the performance improvement potential of careful mapping. Alternative schemes to implement careful mapping are described and compared via trace-driven simulation. The results indicate that careful page mapping can significantly reduce the frequency of cache misses and is an important technique for high-performance memory management systems with real-indexed caches.

1. Introduction

Most general purpose computer systems support both virtual and real (physical) address spaces so as to facilitate efficient memory management policies and to maintain the appearance of large memory availability. The translation from virtual addresses to real memory locations provided by many paged virtual memories allows for considerable flexibility in the actual correspondence of the memory space of executing processes to the physical memory available in the system.

CPU caches are essential in many high-performance computing systems since they provide the access times of a small, fast memory while the storage capabilities of the slower, larger main memories is retained. With both virtual and real address spaces available, CPU cache designers have the option of supporting either virtual-

† The material presented here is based on research supported in part by the National Science Foundation's Presidential Young Investigator and Computer and Computation Research Programs under grants MIPS-8957278 and CCR-8902536, A.T.&T. Bell Laboratories, Digital Equipment Corporation, Texas Instruments, and the graduate school at the University of Wisconsin--Madison. R. E. Kessler has been supported by a summer internship from Digital Equipment Corporation and by graduate fellowships from the National Science Foundation and the University of Wisconsin-Madison.

indexed or real-indexed caches. In a virtually-indexed cache memory locations are associatively mapped to physical cache frames using virtual addresses; with a real-indexed cache the locations are associatively mapped using the real addresses of the locations.

Even though virtual-indexed caches can have faster access times than real-indexed caches (since virtual to real address translation is not required), real-indexed caches are more common [SMIT82]. Virtually-indexed caches suffer from the synonym problem: multiple virtual addresses which should reference the same memory can reside in different cache locations. Consistency problems can also result when I/O devices or cache coherence mechanisms access the memory locations contained in the cache, both of which normally use physical addresses. Though these problems can be solved (see [GOOD87, HIEL86]), their severity can be a determining factor in the choice of real over virtually-indexed caches.

Real-indexed caches are especially appropriate in secondary caches of a system with hierarchically organized CPU caches. Secondary caches service the misses of primary caches which directly service processor memory references. Virtual address translation may be completed in parallel with primary cache access [WABL89] (or may be done quickly since the TLB slice is all that is required [TADF90]), making real-indexed caches as fast as virtual. Even when virtual address translation time must be included in the secondary cache access time, it may not be too large of a burden since secondary caches are accessed less frequently than primary caches.

1.1. Cache Mapping Improvements

The mapping from pages in the virtual address space to page frames in the real address space is often fully associative: a virtual page can be mapped to any one of the page frames in the physical main memory. The cache mapping from blocks to block frames is sometimes direct-mapped: a block can reside in only one block frame. Set-associative mappings provide a spectrum of associativities ranging from the unrestrictive fully associative to the most restrictive direct-mapping. Whereas the virtual page to real page frame mappings are near the fully associative end of the spectrum, CPU cache mappings are usually near the opposite end due to implementation costs and speed restrictions.

For large caches, the disparity between the unrestricted page map and a real-indexed CPU cache map implies that both maps interact to determine the location where a block will reside in the CPU cache. Figure 1 indicates address translation stages used to map a virtually addressed memory location into a real-indexed cache. The virtual address is first translated into a real address. The real address of the memory location is then used to index into the cache. Any of the block frames in the set chosen by the *set* bits of the real address can contain the block. The *tag* bits are used to differentiate among the blocks which map to a set.

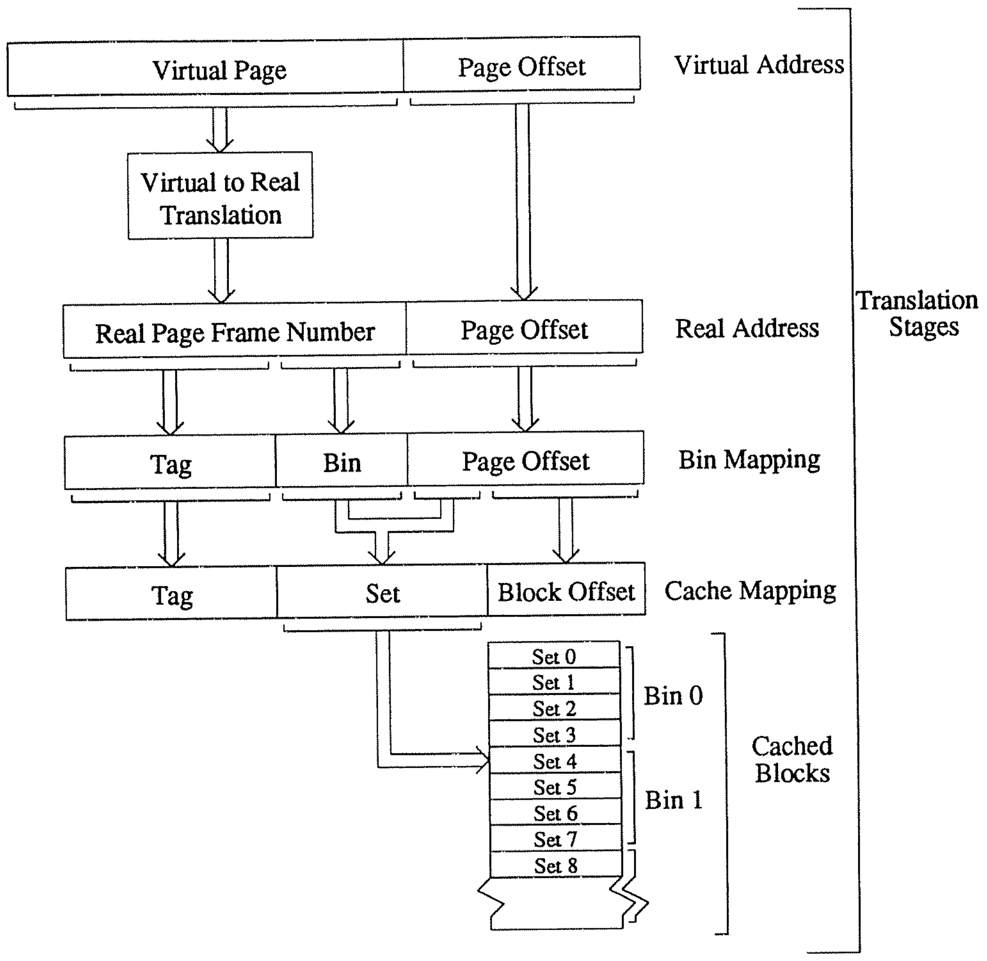


Figure 1. Cache Mapping Overlap with Page Mapping.

This figure shows translation stages which map a virtually addressed memory location into a large (its size divided by its associativity is larger than the page size) set-associative real-indexed CPU cache. The virtual to real page mapping together with the cache mapping determine the cached location of a block. Since the bottom bits of the real page frame number are used to choose the cache set, the behavior of the real-indexed cache mapping is dependent on the virtual to real address translation. Only bit-selection cache mappings [SMIT82] with block sizes not greater than the page size are considered.

Figure 1 shows that the location of a cached page depends both on the page map and the cache map. A cache *bin*[1] is a contiguous sequence of cache sets to which the contents of a page frame will map in a real-indexed cache. Bins are created by the overlap of the *page frame number* and *set* fields. The bin is chosen by the bottom

1. Note that the *superset* of Goodman [GOOD87] is not the same as a cache bin. Pieces of a superset span across *all* bins.

bits of the page frame number, whose value depends on the virtual to real page map. Bins which consist of four cache sets are depicted in Figure 1. Note that the bins properly partition the cache sets. A page frame is said to be in a cache bin when it is one of the page frames which map to the bin. A (virtual) page is said to map to a bin in a real-indexed cache when it is mapped to a page frame (by the virtual to real page map) which is in the bin.

Since the set-associative mapping into a large real-indexed cache depends on the virtual to real page map, a poor page mapping choice can result in a higher frequency of cache misses. Figure 2 shows two possible mappings of the code, data, and stack pages of an address space into real-indexed cache bins. The left portion of the figure shows a mapping where some of the bins are utilized by many pages while others are unused, a poor mapping of this address space, particularly in a direct-mapped cache. The unused bins could be used to eliminate some of the cache misses from the overutilized bins.

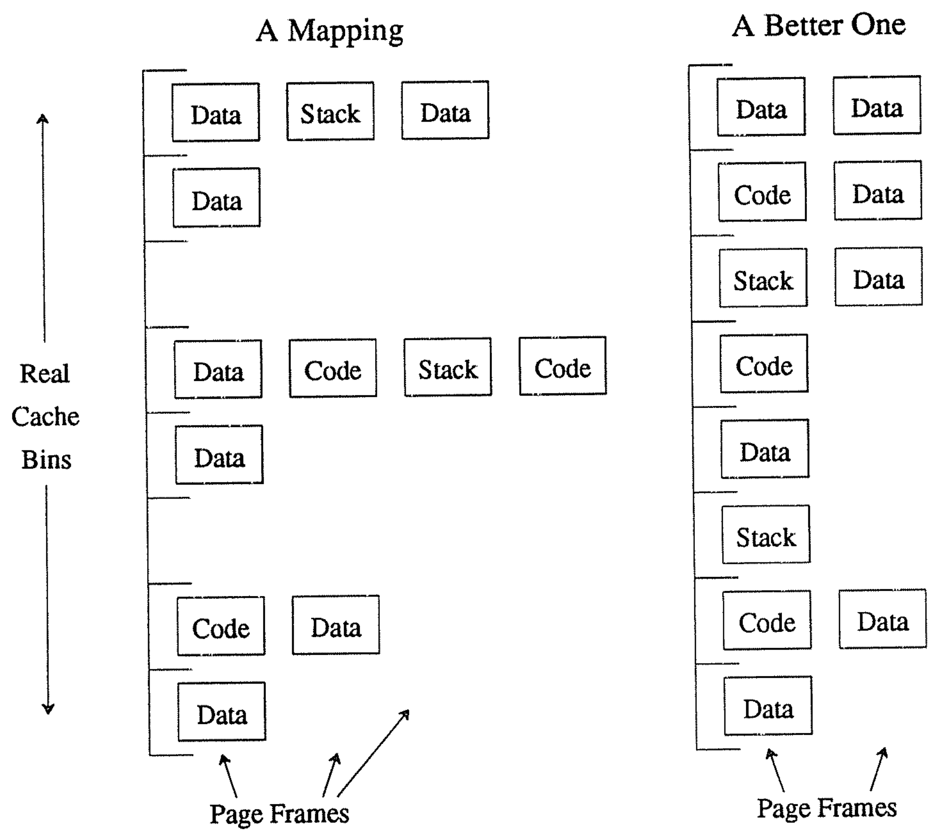


Figure 2. Two Real-Indexed Cache Mappings.

This figure shows two mappings of the code, data, and stack pages of an address space into a real-indexed cache. Page frames can map to arbitrary locations in the cache. This results in some cache bins being under or over utilized in the left mapping, where the utilization of a bin is indicated by the number of pages in the bin. The right mapping is better since the cache bins are evenly utilized.

The right portion of Figure 2 shows a better mapping of page frames into the real-indexed cache. This mapping would likely result in a lower frequency of misses for the application than would the left mapping since the cache bins are more evenly utilized.

The purpose of this paper is to examine virtual page to real page frame mappings which improve overall memory system performance by eliminating real-indexed cache misses. These mappings should be more like the right mapping of Figure 2 than the left mapping. In order to produce good maps, the memory management subsystem of an operating system should consider the differences among page frames, rather than just considering the page frames as a single equivalence class of usable memory, particularly when there is the flexibility to choose among alternate page frames to map a virtual page. A virtual page may only be mapped to a real page once during the lifetime of the page. A poor mapping choice can result in poor performance each the time the mapped memory is referenced. It may be worthwhile to expend effort during the mapping of a page in order to ensure good performance throughout the usage of the page.

1.2. Previous Work

Many studies have considered the usefulness of optimizing program behavior for virtual memory. Ferrari [FERR76] surveys many schemes to reduce the number of page faults by a program executing on a virtual memory system. The placement of data in virtual memory can be modified so that programs will reference few pages at the same time, resulting in fewer page faults. Stamos tries similarly to cluster objects and improve virtual memory performance [STAM84]. Regarding CPU Caches, McFarling [MCFA89] and Hwu and Chang [HWUC89] introduce schemes to scatter commonly used instructions across direct-mapped caches in order to reduce instruction cache misses and result in better memory system performance. Whereas all these program behavior optimizations involve compiler (or user) modifications, the optimizations introduced in this study involve modifications to the operating system memory management software. Though the behavior of the virtual memory references of the application can not be modified by the memory management subsystem, physical memory reference behavior can.

Most CPU cache performance studies of cache performance do not differentiate between virtually-indexed and real-indexed caches. An exception is the study by Sites and Agarwal [SITA88] in which the simulated real-indexed caches perform worse than corresponding virtually-indexed caches (except in cases of frequent context switching) on the same workloads.

A few current systems have been designed with the impact of the virtual to real page mapping on cache performance in mind. In the Sun 4/110, instruction pages and data pages [KELL90] are mapped to even and odd page frames to partition instruction and data entries in its SCRAM cache [GOOC84], giving some of the advantages of split instruction and data caches. MIPS uses a variant of *Page Coloring*, described in Section 3, to improve and stabilize its real-indexed cache performance [TADF90].

1.3. Contributions of This Work

Previous CPU cache performance studies have ignored the effect of operating system memory management policies on cache performance. The performance of real-indexed caches is studied in this work assuming that operating system policies may be modified to reduce the frequency of cache misses. It will be shown that changes to the page mapping can substantially reduce the frequency of misses in real-indexed caches, resulting in better memory system performance.

Various techniques to carefully produce good page maps are discussed in this study, ranging from very simple heuristic functions to more complex ones. The real-indexed cache performance resulting from the mappings produced by these heuristics is compared over a range of large cache configurations. Important properties of each of the heuristics are considered. Operating system design issues will be explored so as to develop high-performance page mapping policies suitable for practical system implementations.

The potential for improvement in real-indexed cache mappings is examined through a simple model developed in Section 2. In Section 3, page mapping policies to reduce the frequency of misses in real-indexed caches are described. Through trace-driven simulation in Section 4, these page mapping policies will be shown to have substantial effects on the performance of large, real-indexed caches. Furthermore, the simple model will be validated.

2. A Simple Page Conflict Analysis

A simple model is developed in this section to gauge the potential benefits of a virtual to real page mapping algorithm tuned to increase the performance of a real-indexed set-associative cache. The metric used by the model is *conflicts*, C, the number of page frames that can conflict with page frames already in the cache. A conflict occurs when more than A page frames map to a given cache bin, where A is the associativity of the cache; the number of conflicts for this bin is calculated as $u-A$ where u is the number of application pages that map to the bin. The correlation between page frame conflicts and cache conflicts (misses) is positive, so conflicts is a reasonable measure of the goodness of a given cache mapping, as will be shown in Section 4.2.2. Assuming a direct-mapped cache, the number of conflicts in the left mapping of Figure 2 is 6 and the right mapping of the same figure is 4. Similarly, there are 3 and 0 conflicts in the mappings, respectively, for a 2-way set-associative cache.

The U application pages are mapped to page frames by sampling (without replacement!) from a finite population of page frames. There are P available page frames, and exactly $\frac{1}{B}P$ (an integer number) page frames are assumed to belong in each of the B bins. To calculate the average number of conflicts from a random mapping, the probability that exactly u of the U referenced application pages map to a page frame in a given bin is calculated as a hypergeometric distribution [MILF77]:

$$P(u \text{ in bin}) = \frac{\binom{\frac{1}{B}P}{u}\binom{\frac{(B-1)}{B}P}{U-u}}{\binom{P}{U}}. \tag{1}$$

The binomial coefficient

$$\binom{a}{b} = \frac{a!}{(a-b)!\,b!}$$

represents the number of ways to choose b elements from a elements (without replacement).

The numerator in Equation 1 represents the number of ways in which exactly u application pages map to the given bin. It is the product of the number of ways that u out of the possible $\frac{1}{B}P$ page frames from the bin can be chosen and the number of ways the other $U-u$ application pages can be chosen from the other $\frac{(B-1)}{B}P$ available page frames. The denominator in Equation 1 represents the number of different ways the U application pages can map to the P available page frames. The ratio of these terms gives the required probability that exactly u application pages map to the given bin.

The expected number of conflicts of a mapping of U pages can be calculated using Equation 1 as:

$$C_{avg} = B \sum_{u=A+1}^{min(\frac{P}{B},\,U)} (u-A)P(u \text{ in bin}) \tag{2}$$

by multiplying $u-A$, the conflict page frames, times the probability of the occurrence of u page frames in a bin to produce the expected number of conflict page frames in each bin. This is then multiplied by the number of bins, B, to produce the expected number of conflict page frames.

One can also calculate the bounds on C. Figure 3 plots C_{avg} and its bounds for various application sizes. $C_{\max}$ is almost linear with the application size starting from the origin, $U=0$. The worst case occurs when the pages map to the minimum number of bins (probably one). $C_{\min}$ remains equal to zero up to the point where the application size equals the cache size ($U = N$), then the number of conflicts increases linearly with the application size. The best case corresponds to even utilization of the cache and thus results in no conflicts until $U > N$. Since C_{avg} must be between $C_{\min}$ and $C_{\max}$, it also increases nearly linearly as U increases. It tends to remain closer to $C_{\min}$ than $C_{\max}$, particularly for small and large values of U.

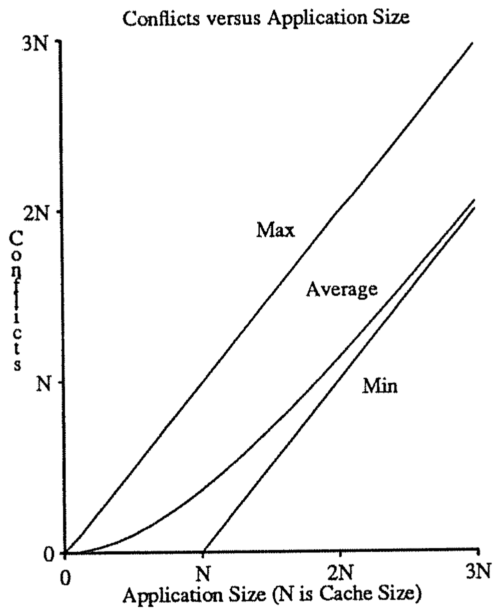


Figure 3. Cache Mapping Conflicts.

This figure indicates C_{avg} (as calculated by Equation 2), $C_{\max}$, and $C_{\min}$ for various application sizes (U). The modeled system is a 1 megabyte direct-mapped cache ($N = 64$) backed up by a 128 megabyte main memory, with a page size of 16 kilobytes.

Figure 4 illustrates the the difference between C_{avg} and $C_{\min}$ more dramatically. It plots the conflicts due to the random mapping, the mapping conflicts ($C_{avg} - C_{\min}$), for varying application sizes and multiple associativities. The mapping conflicts are an indicator of the potential cache miss reductions which could result from careful virtual page to real page frame mappings. While a naive page mapping function would result in C_{avg} conflicts, a careful page mapping function could result in close to $C_{\min}$ conflicts in a given mapping. The mapping conflicts, $C_{avg} - C_{\min}$, then indicates the potential savings of a careful mapping function.

Figure 4 indicates that the mapping conflicts are maximized at the point where the application size equals the cache size. There are few mapping conflicts for application sizes which are small relative to the cache size since the probability of many collisions among few page frames is small. $C_{avg} - C_{\min}$ increases as the application

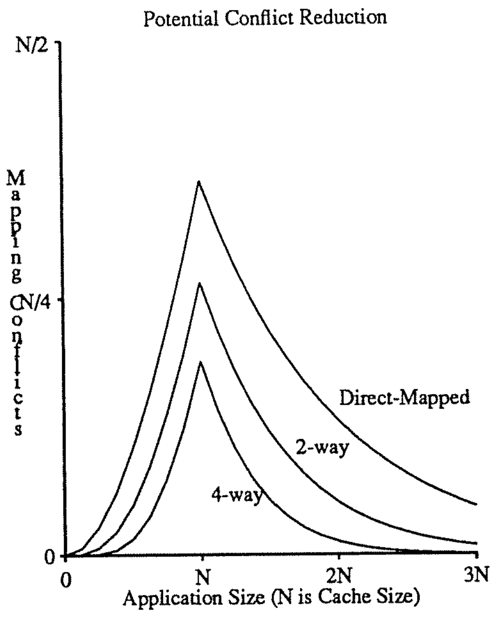


Figure 4. Random Mapping Conflicts.

This figure shows the difference between the conflicts due to a random mapping and the minimum number of conflicts, the mapping conflicts ($C_{avg} - C_{\min}$), for different application sizes and cache associativities. The configuration is a 128 megabyte main memory in 16 kilobyte pages with a 1 megabyte cache ($N = 64$).

size approaches the cache size since the C_{avg} increases while $C_{\min}$ remains at 0. The mapping conflicts approach 0 as the application size increases beyond the cache size since $C_{\min}$ increases linearly while C_{avg} increases at a somewhat slower rate. $C_{avg} - C_{\min}$ is small for large application sizes since the cache is likely to be highly utilized regardless of the mapping scheme, and thus will result in close to the same number of conflicts. An inference from this data is that *the largest gain from careful mapping will come when the application size is approximately the cache size.* Other application sizes will also benefit from careful mapping, but this is the best case. When the cache is underutilized or highly utilized, careful mapping becomes less useful and necessary.

Figure 4 also shows that $C_{avg} - C_{\min}$ decreases as the associativity of the cache increases. This is not surprising since, as an extreme case, there will be no mapping conflicts in a fully associative cache, regardless of U. *There is less need for careful mapping when the cache is associative.* Careful mapping will be most useful with a direct-mapped cache.

3. Implementing Careful Mapping

While the previous section attempted to gauge the potential real-indexed cache performance improvement of careful virtual to real page mapping, it did not determine if the potentials were achievable. In this section, careful page mapping implementations integrating important memory system aspects are presented. First, the goals and implementation considerations of careful page mapping are outlined. Then implementation alternatives are described.

3.1. Careful Mapping Goals and Assumptions

The purpose of a page mapping function is, given a virtual page, to find a page frame for use in mapping a page. In choosing a location (page frame) to which the virtual page maps in main memory, the page mapping function is also (implicitly) choosing the bin (location) to which the virtual page will map in the cache. Choosing a page mapping function which produces good page maps is important to overall memory system performance optimization. Considerations in the design of page mapping functions are discussed in this section.

3.1.1. Relationship of Page Mapping to Page Replacement

The mapping functions assume the availability of a pool of alternative page frames when mapping a virtual page. Page mapping decisions are strictly performance optimization policies. A pool of available pages is not required to ensure the correctness of the page mapping functions, only to improve their performance. The more page frames that are available for mapping, the more freedom the page mapping function has to improve the mapping and thus improve CPU cache performance.

When main memory is underutilized, many page frames will naturally be available for page mapping. In that case, the page mapping policy is independent of the *page replacement policy* since no page frames need be replaced. However, when main memory is scarce, the page mapping policy is intricately related to the page replacement policy. In steady state, it is either page replacement or the freeing of a page (due, for example, to the execution completion of a process) which makes page frames available for mapping.

Careful page mapping is only one of several motivations for having an available pool of page frames. Such a pool can smooth out large variations in memory demand, allowing the system to reduce response times during periods of peak memory demands while not substantially increasing the activity to virtual (disk) storage since the pages can be *reclaimed* while still in main memory [BABJ81]. A free page frame pool can be maintained regardless of the particular page replacement policy in use, replacement can occur often enough to maintain the size of the pool, and its size can be tuned to optimize overall performance: too large of a pool may reduce virtual memory performance (by increasing the page fault frequency), and too small of a pool may increase CPU cache misses. It is preferable for a page mapping policy to adjust to all situations, including when the pool of available page frames is small.

For analysis, it is essential to consider the page mapping policy to be independent of the page replacement policy. It is infeasible to analyze all page replacement policies that have been proposed, analyzed, or implemented. Smith's bibliography [SMIT78] includes references to some of the possibilities, and even more are available. In this study, the implementation of page mapping functions is considered independent of the page replacement algorithm.

3.1.2. The Importance of Page Conflict Minimization

The page mapping functions attempt to minimize the (static) page conflicts of a given mapping into a unified[2] cache: a mapping which minimizes conflicts is considered optimal. It is important to keep in mind, however, that minimizing page conflicts should not be the ultimate goal of a mapping. Most importantly, the mapping function should produce mappings that minimize the mean and variance of the execution time of a given program, preferably with a low overhead (in terms of space and time) in producing the mapping. When the frequency of cache misses is reduced, the execution time of a program is reduced, as long as the cost of reducing the miss frequency is no too great. The assumption made in considering careful mapping implementations, one which will examined in Section 4.2.2, is that reducing the number of page conflicts also reduces the frequency of cache misses. Efficient careful mapping functions to eliminate page conflicts are considered in this study.

3.1.3. Multiple Address Spaces

In most systems, there are multiple processes (accessing multiple address spaces) executing at the same time, interleaving their use of the CPU. Multiple, concurrent, address spaces complicate the conflict minimization process. In their presence, there are two (not necessarily conflicting) page conflict minimization goals to satisfy: (1) minimize the conflicts *within* each address space, and (2) minimize the conflicts of *all* address spaces as a whole. The conflicts that span multiple address spaces may have only a small effect on the frequency of misses. Conflicts within an address space not currently in use, perhaps because a process is suspended, for example, would not cause cache misses. With large numbers of memory references between context switches, conflicts

2. Mapping functions to minimize the number of misses in "split" caches are not considered in this study. An example of split caches are split instruction and data caches [SMIT82]. With these caches, the mapping function may want to concurrently minimize the conflicts of instruction pages in the instruction cache while minimizing the conflicts of the data pages in the data cache.

within a single address space can have a dominant effect on cache performance, while conflicts across address spaces will only cause misses in proportion to less frequent context switches. In general, minimizing conflicts within a single address space is considered most important in this study, minimizing conflicts across address spaces is a secondary consideration.

In addition to handling performance optimization with independent address spaces, a careful page mapping function should adapt to sharing among the address spaces. Pages shared among different address spaces complicate the conflict minimization process since their mapping effects more than one address space. A shared page would preferably be placed in a page frame which minimizes the conflicts in all address spaces sharing the page.

3.1.4. Cache Size Independence

Rather than optimizing for a single cache configuration, it is desirable that a page mapping function produce mappings to concurrently minimize page conflicts in a variety of cache configurations. This way, mappings produced for one cache configuration are also optimized for other configurations.

Size independence would simplify software management. For example, it would allow the same operating system executable to be used on machines with different cache configurations, while eliminating conflicts in each case. Another important reason to produce mappings which are cache size independent is that it allows conflicts to be minimized in multiple caches *simultaneously*. This is very important with a hierarchy of real-indexed CPU caches. A size independent mapping function could simultaneously reduce the number of conflicts in all caches in the hierarchy. This is essential for optimum overall CPU cache performance. In addition to these implementation advantages, size independence also allows a simulation advantage. Multiple cache configurations can be simultaneously simulated with the same heuristic function. One of the implementation alternatives considered in this study, the Hierarchical Heuristic, produces mappings which are size independent. The usefulness of size independence is demonstrated in the trace-driven simulation section of this study.

3.2. Careful Mapping Implementation Alternatives

The task of finding the performance-optimal page mapping policy is a difficult one. As is often the case with difficult problems, *heuristic* solutions are appropriate. The heuristics correspond to simple optimization goals which are extracted from a larger, more complicated optimization problem. A simple heuristic may be preferred over a more complex one, unless the more complex heuristic consistently makes superior decisions. With these considerations in mind, two simple and two more complex heuristic algorithms will be discussed in this section, *Page Coloring*, *Bin Hopping*, *Sequential*, and *Hierarchical*, respectively.

3.2.1. The Page Coloring Heuristic

The mapping of a virtual address space into a virtually-indexed cache has the desirable property of even cache utilization by the memory locations of the address space. Successive virtual pages will not conflict in the cache and an application which is smaller than the cache size will often map into the cache without conflict. The most straight-forward way for a mapping into a real-indexed cache to appear like a virtually-indexed mapping, and thus obtain the advantages of virtually-indexed caches, is for the bin bits of each virtual page to be equivalent to the bin bits of the page frame it maps to. This technique has been called *page coloring*. A variant of it is used by MIPS [TADF90].

The simplicity of the page coloring heuristic is perhaps its main advantage. No state information is necessary per address space. The task of the page mapping function is simply to try and (equality) match the bin bits of the virtual page to a real page frame. When a page frame is not available in a bin corresponding to the virtual page, any available page frame can be used to map the page, likely the first available. Simplicity is a very useful property of Page Coloring, particularly given the complexity of current memory subsystems.

More sophisticated implementations of Page Coloring are possible and likely to be useful. As the implementation is described, commonly used virtual addresses (like the stack, for instance) from different address spaces will be mapped to the same cache bins, leading to many inter-address space conflicts and cache misses. Rather than requiring an exact equality between the bin bits of the virtual page and the page frame, the equality could instead be offset by a different constant factor for each address space (or for each contiguous segment) [TADF90]. As long as the offset factor were different across different address spaces, commonly used areas of each virtual address space would not map to the same cache bins and these inter-address space conflicts would be reduced. Two versions of Page Coloring were chosen for consideration in this study: (1) an exact match of bin bits between the virtual page and real page frame, and (2) a match of bin bits exclusive-ored with a different PID for each address space to ensure that commonly used virtual addresses do not map to the same spot. The precise form of Page Coloring used by MIPS was not known.

3.2.2. The Bin Hopping Heuristic

Another simple way to ensure even cache bin utilization is to map application pages to page frames in successive bins. This Bin Hopping is pictured in Figure 5. Mapping pages to successive bins ensures that conflicts will not occur unless the application size is larger than the cache size. It also exploits temporal locality in that pages which are mapped at the same time will tend to be placed so that they do not conflict in the cache. Assuming these pages will tend to be referenced at the same time, pages which are referenced together will likely be placed in separate cache bins. Bin Hopping can be implemented by maintaining a bin pointer that is initialized to a random value and incremented at the time each page is mapped.

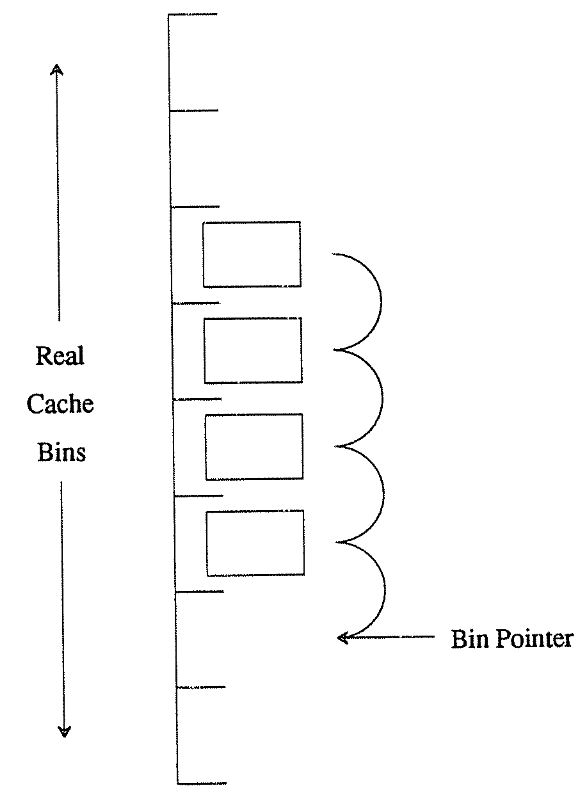


Figure 5. Page Mapping By Bin Hopping.

This figure pictures a page mapping which could occur with the Bin Hopping heuristic algorithm. Pages are mapped to page frames which map to successive cache bins.

The implementation of Bin Hopping is straight-forward. The bins can be traversed successively with the bin pointer until one is found with an available page frame, saving the new bin pointer after it is used. The simplicity of the Bin Hopping heuristic, like Page Coloring, is its largest asset. A very small amount of state information needs to be maintained per address space, just the bin pointer.

3.2.3. The Sequential Heuristic

One of the strengths of the Page Coloring and Bin Hopping heuristics is also one of their weaknesses. The small amount of state information used limits the storage space required, but can reduce the effectiveness of the mapping function. When there are not available pages in each bin, these heuristics can have problems producing a good mapping. Information indicating the number of pages previously mapped to each cache bin by each address space could be used to advantage by a heuristic function. Of course, this mapping information is obtainable by examining the mapping of virtual pages to page frames, which could be a costly procedure. Since the mapping function would more efficiently execute if the number of pages mapped to each bin were more readily available, an implementation might want to store this information separately.

In searching for the ''best'' bin, the state information used to determine the suitability of a given bin is the pair *<used, free>*, where *used* is the number of pages previously mapped to the bin by the address space, and *free* is the number of page frames available for mapping to this bin. A simple algorithm to map a page to a page frame would be to sequentially look at the *<used, free>* pairs for each bin and choose the bin which best meets the standards of suitability. This algorithm is called the *Sequential* heuristic. The Sequential heuristic maintains system-wide *free* information to indicate the number of available page frames in each bin, just as the Page Coloring and Bin Hopping heuristics need. The added state of the Sequential Heuristic is the *used* information per active address space (process).

The heuristic determining the ''best'' bin is affected by the goals of the mapping: minimizing intra-address conflicts is considered more important than minimizing inter-address conflicts. The parameters used in this bin ranking are the *<used, free>* pairs for each bin. The specific rules used for the ranking, in priority order are: (1) the bin must have at least one free page frame, (2) the bin should have the fewest pages used by this address space, and (3) the bin should have the most page frames available. Rule (2) minimizes conflicts within the address space while rule (3) minimizes conflicts between different address spaces since more available page frames implies less of them are used. If these rules fail to produce a preferable bin, an arbitrary choice can be made, such as a random choice.

An example of a choice made by the Sequential Heuristic is pictured in Figure 6. Of all the *<used, free>* pairs, the <1, 3> bin is considered best since there are several available page frames and only one previously mapped page frame. After choosing the <1, 3> bin, the page mapping function would extract one of the available page frames from the bin and map the virtual page to the real page frame. The <1, 3> pair would be modified to <2, 2> to indicate the changed system state.

3.2.4. The Hierarchical Heuristic

The $O(B)$ time computational complexity required by the Sequential Heuristic to sequentially traverse the bins could be too expensive when there is a large number of bins. Alternatively, a bin can be chosen by the Hierarchical Heuristic with the aid of a tree structure called a bin tree. The time complexity of the Hierarchical Heuristic is $O(\log_2 B)$, considerably smaller than $O(B)$ for large values of B. The cost of this time reduction is

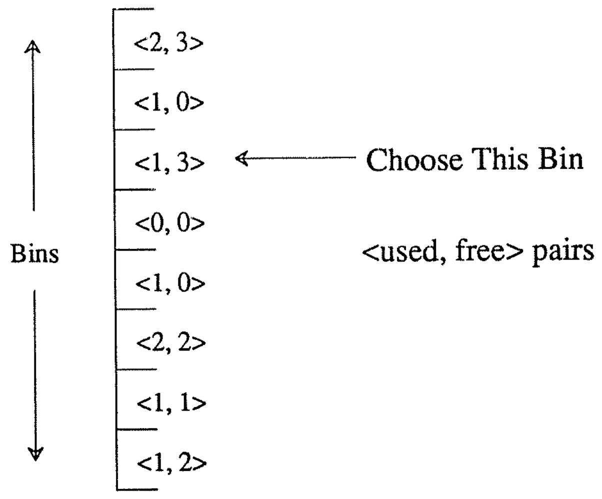


Figure 6. Page Mapping by the Sequential Heuristic.

This figure shows the bin which would be chosen as then best bin by the Sequential Heuristic. Pictured are the *<used, free>* pairs for each bin. The heuristic will look at all bins and determine that the <1, 3> bin is best.

that the Hierarchical Heuristic may make an inferior decision to the Sequential Heuristic for a given cache, though in most cases it will produce a good mapping.

In addition to its time complexity benefit, the Hierarchical Heuristic produces mappings that are cache size independent. With the proper implementation, decisions made at the higher levels of the tree correspond to those needed when mapping into a smaller cache. Decisions made at lower tree levels only refine those made at upper levels and correspond to those needed when mapping into a larger cache. This will be subsequently shown.

A bin tree is a fully balanced binary tree with a number of leaf nodes equivalent to the number of bins in the cache; each leaf is associated with exactly one bin. The value of a leaf node is the number of page frames in the corresponding bin. The value of interior nodes is the sum of the values of its children. Separate bin trees are used to maintain the used and free parameters corresponding to each bin. The value of the root node of a used (free) bin tree is the number of used (free) page frames. The values at each node are used to determine the direction when the tree is being traversed during the mapping of a page. A single free bin tree can be used system-wide, but a used bin tree is needed by each address space (process), just as with the bin arrays of the Sequential Heuristic. The use of binary trees is assumed throughout this study, higher branching factors could also be used but may not maintain size independence for all cache configurations.

When a new virtual page is mapped to a page frame, the bin trees are traversed from the root to one of the bins (leaves) at the bottom of the trees. During a traversal, exactly $\log_2(B)$ choices among alternative subtrees determines the particular traversal path through the tree from the root to the leaf corresponding to a bin. The key determinant of the goodness of the Hierarchical Heuristic is the goodness of these decisions. The decision at each node is made based on the *<used, free>* pairs corresponding to the two children of the node. The traversal direction is determined by precisely the bin ranking heuristic function used by the Sequential Heuristic. The *<used, free>* values passed to the bin ranking function are intermediate tree values rather than the final bin values as with the Sequential heuristic. The same ranking function can be used by both since the goals of the decision are the same in each case.

Figure 7 shows an application of the direction heuristic to a simple example. The value of each node in the *used* and *free* trees are given within in the nodes. Traversal starts from the root of the bin trees. From there, a decision is made based on the <3, 6> pair corresponding to the left subtree and the <3, 3> pair of the right subtree. Since both subtrees have available page frames and the number used from each is equivalent, a decision to traverse the left subtree is made since it has more available page frames. A decision must then be made based on the <2, 5> and <1, 1> pairs of the nodes at the next level. The right subtree is chosen for traversal since less page frames have been mapped to it. Bin tree traversal is then complete since a leaf node is reached. To update the state to account for the page mapping, the values of the traversed nodes would be modified to <7, 8>, <4, 5>, and <2, 0>, respectively, from top to bottom.

Figure 8 illustrates the labeling of tree nodes in a bit-reversed manor so that the Hierarchical Heuristic produces size independent mappings. Using the Hierarchical Heuristic with alternative labeling schemes may not produce size independent mappings. The labels of the tree nodes are a bit-reversal of the bit strings obtained from an increasing (binary) ordering of the nodes on a level from left to right. By definition, the value of a leaf node is the number of *used (free)* page frames whose bottom bits match the leaf node label. With this labeling, it can further be shown that the value of *all* tree nodes is the number of page frames whose bottom bits match the label of the node, given that a parent node value is the sum of its children values.

The low-order (right) digit of the nodes on the left half of the bin tree are all 0 while the corresponding bit is always 1 for the nodes on the right half. Since the first decision made by the Hierarchical Heuristic during mapping bin tree traversal is to further traverse either the right or left half of the tree, this decision corresponds to the choice of the bottom bit of the page frame to map the page. This is exactly the conflict elimination decision required for a cache with 2 bins, ensuring that the Hierarchical Heuristic reduces the conflicts in a cache with 2

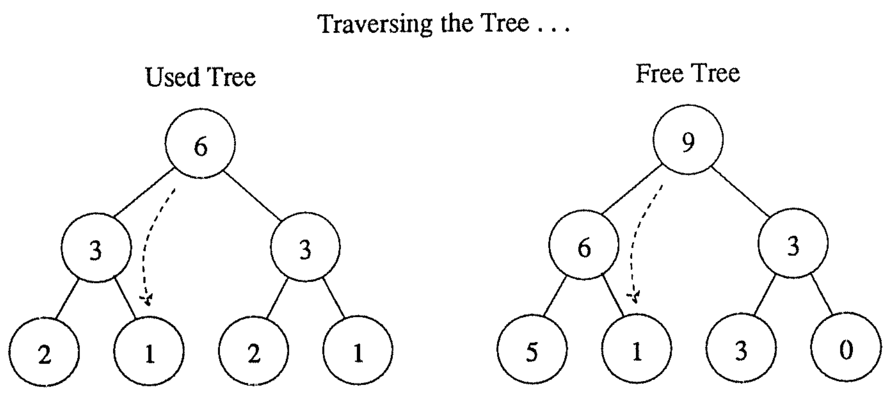


Figure 7. Bin Tree Traversal.

This figure shows an example of the choice of a bin to map a virtual page by used and free bin tree traversal with the Hierarchical Heuristic. The values of the tree nodes are given. The values of the leaves of the used bin tree are the number of pages already mapped to the corresponding bin. The leaves of the free bin tree represent the number of page frames available for replacement in the corresponding bin.

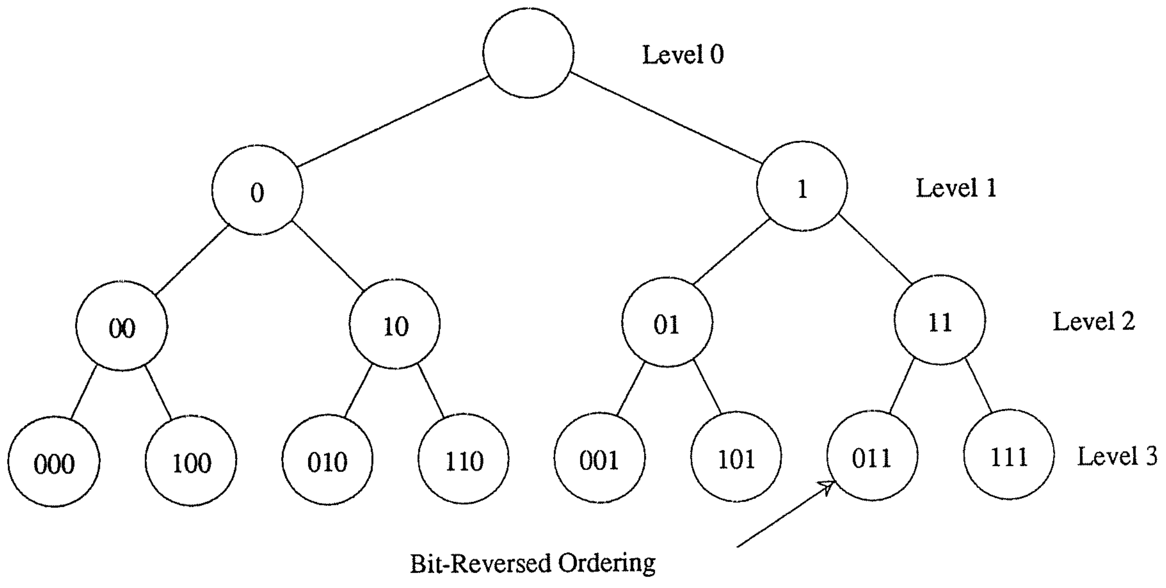


Figure 8. Mapping Tree Nodes to Bins.

This figure shows a relationship of bin numbers to tree nodes which allows the Hierarchical Heuristic to produce size independent mappings. It is the bit-reversal of numbering the nodes with labels in increasing order from left to right.

bins even though the tree in Figure 8 is specifically targeted for a cache with 8 bins (since there are 8 leaves). The essence of cache size independence is that in eliminating conflicts in a cache with more bins, conflicts are automatically eliminated in caches with less bins, as illustrated by this example. A more formal proof that the Hierarchical Heuristic produces size independent mappings has been developed [KESS90]. Space limitations preclude its presentation here.

4. Trace-Driven Simulation Performance Analysis

This section describes results of trace-driven simulation of real-indexed caches when using the heuristics described in the previous section. Trace-driven simulation allows the usefulness of the careful page mapping heuristics to be measured. While the simple analytic model outlined in Section 2 provided estimates of page conflicts and potential conflict reductions that could occur with careful mapping into a real-indexed cache, the frequency of cache block misses determine the goodness of a cache mapping. Trace-driven simulation can provide an accurate measure of cache performance under the traced workloads.

4.1. Methodology

4.1.1. The Traces

The traces used in the study were collected at DEC Western Research Laboratory [BOKL89, BOKW90] and are more completely documented elsewhere [KESS90]. Each trace consists of the execution of 3 to 6 *billion* instructions of very large workloads on a load/store architecture, referencing from 8 to over 100 megabytes over the length of the traces. The length of these traces is sufficient to overcome the cold-start intervals of even these large caches.

The traces of the multiprogrammed workloads represent the actual execution interleaving of the processes on the traced system. The **Mult2** trace includes a series of compiles, a printed circuit board router, a VLSI design rule checker, and a series of simple UNIX commands, all executing in parallel (approximately 40 megabytes active at any time) with an average of 134,000 instructions executed between each process switch. The **Mult2.2** trace is the Mult2 workload with a switch interval of 214,000 instructions. The **Mult1** trace includes the processes in the Mult2 trace plus an execution of the system loader (the last phase of compilation) and a Scheme (Lisp variant) program (75 megabytes active) and has a switch interval of 138,000 instructions. The **Mult1.2** trace is the Mult1 workload with a switch interval of 195,000 instructions. **Sor** is a uniprocessor successive over-relaxation algorithm which uses very large, sparse matrices (62 megabytes). The **Tv** trace is of a VLSI timing verifier (96 megabytes). **Tree** is Scheme program which searches a large tree data structure (64 megabytes). **Lin** is a power supply analyzer working on a register file with sparse matrices (57 megabytes).

4.1.2. The Simulator

Both hardware and software system components are simulated in order to determine the effects of operating system policies on real-indexed cache performance. Previous trace-driven simulation results analyzing cache performance have assumed that operating system policies are fixed and independent of the studied cache configurations. This assumption is removed in the results presented in this paper. Since different virtual page to physical page frame mapping policies are studied, the simulator incorporates memory management policies of the operating system along with hardware cache management implementations. This gives the simulator the capability to analyze the interaction of hardware and software system components.

A global least-recently-used (LRU) page replacement policy is used in this study. During the simulations, an exact ordering of the page frames from most recently used to least recently used is maintained (an LRU list). A virtual page is mapped to a corresponding page frame at the point when the page is first referenced. Though the implementation of an exact LRU replacement policy may be infeasible in practice, it represents a replacement policy with desirable properties which is feasible to implement in a simulation environment. The page frame at the bottom of the LRU list, the least recently used one, would normally be the best candidate page frame for use in the mapping.

Rather than requiring that the last page on the LRU list must be chosen for mapping, the careful page mapping policies are given the freedom to choose among a number of page frames at the bottom of the LRU list. This is shown in Figure 9. That is, the pool of available page frames are those at the bottom of the LRU list. During each execution of the page mapping function in the simulations of this study, exactly the same number of page frames are available for mapping, even at the beginning of the simulation when the entire main memory is considered unused. The default available page frame pool used in this study is 4 megabytes of the 128 megabytes of main memory available in 16 kilobyte pages.

Before each real-indexed cache simulation is started, the LRU list is initialized in a different random order. This is intended to reflect that after a system has been executing for some time, page frames are no longer in any meaningful order. This will likely be the case in a system which does not differentiate among its page frames; eventually the page frames get jumbled into an arbitrary order. Given this random ordering, the mapping algorithms have a random sampling of page frames available for mapping at the beginning of a simulation. As execution continues, the available page frames are determined by the previous mapping choices and the memory reference pattern of the traces.

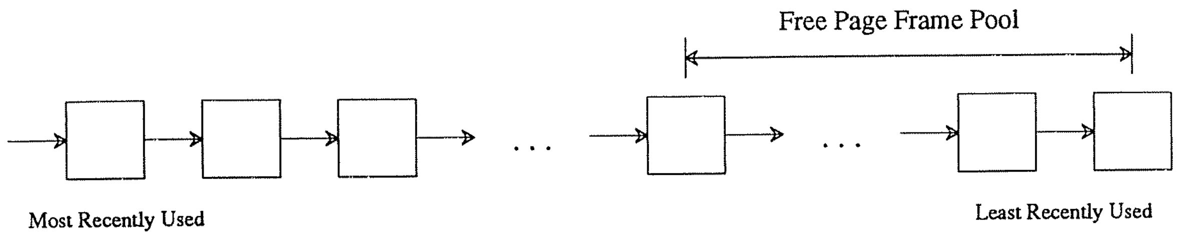


Figure 9. LRU List Implementation With Available Page Frames.

This figure shows the global LRU list of page frames and the free page frame pool. An exact list of page frames from most recently used to least recently used is maintained during real-indexed cache simulations. When a virtual page is mapped to a page frame, the bottom page frames on the list are made available for mapping.

The CPU cache configuration under consideration is is a two-level hierarchical system. The primary (level one) caches are split direct-mapped write-back[3] instruction and data caches backed up by a unified secondary (level two) write-back cache with a random replacement policy and 128 byte blocks. The primary caches are split instruction and data caches of 32 kilobytes each with 32 byte blocks. Different secondary cache configurations are simulated. The simulations in this paper are of multi-level caching systems which do not maintain inclusion [BAEW88]. Cold-start simulation results are given.

4.1.3. Randomness in The Simulator

Trace-driven simulation results are often presented without indication that there is randomness in the results from a given trace. Although results vary across traces of different workloads, performance metrics from a trace of a given workload are presented as exact, non-varying, values. This presentation of cache performance results is appropriate if the process of analyzing the trace data is entirely deterministic. Trace-driven simulations are not always deterministic, however. Simulations of caches with truly random replacement policies are inherently non-deterministic, though the variations across different simulations are likely to be small.

In real computer systems there exists more potential for variability in cache performance than captured by a given trace. Not only will different workloads result in different cache performance, multiple executions of the same workload can also behave differently. The sources of system variability are eliminated within a single trace, however, since trace data represents a single execution of a workload on the traced system. Variability is eliminated in real address traces since a single virtual to real page mapping has been used when gathering the trace

3. On a write miss, space is allocated and the cache block is read in (write-allocate). On a miss to the data cache which requires a write of a dirty block, the write-back is executed followed by the block read.

data.

In this simulation study, just as in real systems, the performance of a real-indexed cache under a given workload is a random variable dependent upon the virtual to real page mapping. The traces used in this study are virtual address traces of user program references, some of them are multiprogramming traces. Virtual address traces allow the flexibility to simulate different virtual to real address mappings when determining real-indexed cache performance. Trace-driven simulation results of real-indexed caches vary across different simulations, even those using precisely the same trace data, due to the variability in the virtual to real address mappings used in the simulations. Variability is introduced into a real address trace from a static virtual address trace by changing the mapping to real addresses.

The variability in simulation results is addressed in this study by presenting mean, median, and confidence intervals of random samples. A sample size of 4 has been found to be sufficient in most cases in this study. Larger sample sizes would be preferred, but, the length of the traces makes it difficult to gather a large number of samples. Fortunately, the long traces averaged out much of the randomness, making 4 samples a reasonable number.

Assuming the samples were taken from a normal distribution, the 90% confidence intervals used in this study were can calculated utilizing the Student-t distribution [MILF77]. With this technique, confidence intervals are expressed as an interval about the mean of the samples. The width of the confidence interval is directly proportional to the standard deviation. Though the assumption that the random variables x_i are normally distributed can lead to inaccuracies for small numbers of samples (such as the 4 used here), this technique is a straightforward way to gather confidence in the simulation results.

4.1.4. Miss Ratio Can Be Misleading

Most cache performance studies express results in terms of miss ratios, the number of cache misses divided by the number of cache references. Miss ratio is not used in this study since it can be misleading when simulating a hierarchy of CPU caches. The *local miss ratio* [PRHH89] of the secondary cache (on which the concentration of this study is focused) depends on the frequency of misses from the primary caches. A mapping change which *decreases* the frequency of misses in the primary caches and *decreases* the frequency of misses in the secondary cache would actually *increase* the local miss ratio of the secondary cache if the miss frequency were decreased more in the primary caches than the secondary cache. The increase in local miss ratio is misleading since memory system performance will improve when the miss frequency from all caches is reduced.

Cache performance is measured in this study by the frequency of misses, the misses per simulated instruction (MPI), to avoid these problems. MPI is closely related to the *global miss ratio* of Przybylski et. al. [PRHH89].

4.2. Results

4.2.1. The Usefulness of Careful Page Mapping

Variations in the virtual to real page mapping can result in substantial variations in the performance of real-indexed caches. Figure 10 shows results of several trace-driven simulation runs of different mappings of the Mult2.2 trace. 100 million instruction averages of the misses per instruction (*MPI*) are plotted for different simulations and configurations for the length of the Mult2.2 trace. The results compare the Hierarchical Heuristic mappings to random mappings for real-indexed secondary caches ranging from 1 megabyte (top) to 16 megabytes (bottom). The Hierarchical Heuristic is used in comparing to random mapping in this section due to its good performance and the simulation advantages of the size independent mappings it produces. Careful mapping implementation alternatives are compared in detail in a subsequent section.

Careful mappings are consistently better than random mappings for the Mult2.2 trace. This is shown in Figure 10 since the simulations of the random mapping (dotted lines) produce a higher frequency of misses than does the mappings from the Hierarchical Heuristic (solid lines). The different simulations of careful mappings tend to remain closely together and follow an outline which is largely dictated by the characteristics of the workload. The simulations of random mappings show ''spikes'' in the results caused by page conflicts among heavily used pages which result in bursts of cache misses. A substantial fraction of the misses from the random mappings can be attributed to a poor mapping into the cache, rather than the workload behavior itself. Note that some of the spikes appear in the random mappings of both the one and four megabyte cache simulations. This occurs since the simulations of different cache sizes are of the same virtual to real page mapping, and the results are correlated.

Variability in cache behavior for different virtual to real mappings is undesirable. A substantial range of behaviors are possible between the best and worst mappings of a workload into the caches of the system, resulting in different execution times of the workload. Figure 10 shows evidence that the mappings produced by careful mapping are less variable than the mappings produced randomly, a desirable trait. Even though careful mappings remove much of the variability in cache performance, variability is not entirely removed. In more rare circumstances than with random mapping, a careful mapping implementation may also make a poor decision.

Table 1 shows the end-result statistics of simulations of the Mult2.2 trace, results from the data pictured in Figure 10. It shows confidence intervals (mean ± error) and median of four samples of simulation results from

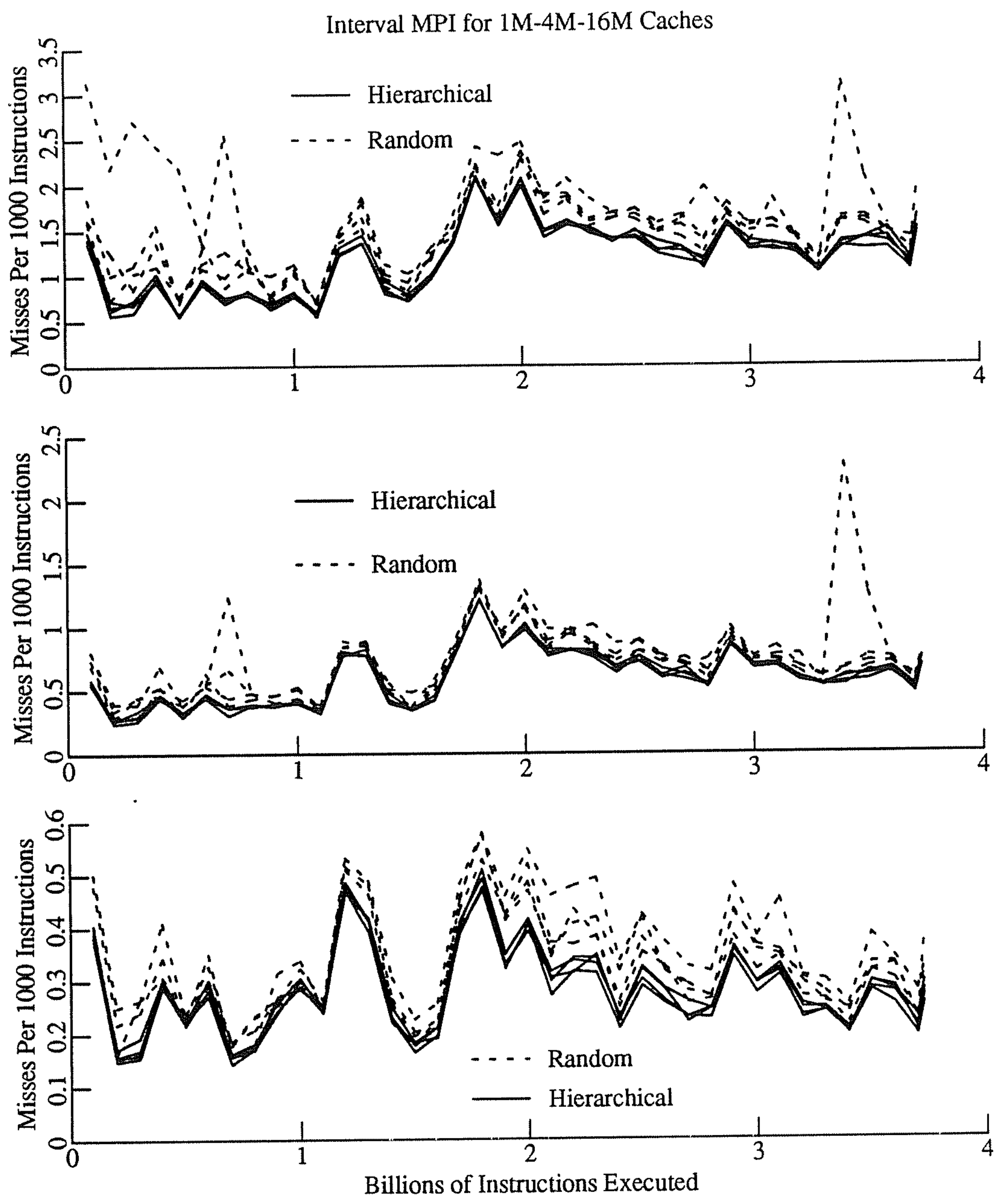


Figure 10. Mult2.2 Mapping Variability Effect.

This figure shows the effect of four different virtual to real mappings generated by the Hierarchical Heuristic (solid lines) and generated randomly (dotted lines) on the misses of three real-indexed secondary caches. The average *MPI* for the previous 100 million instructions of the Mult2.2 trace is shown for 1 megabyte (top), 4 megabyte (middle), and 16 megabyte (bottom) secondary direct-mapped caches. Note that the scales are different for each graph.

The mappings produced by the Hierarchical Heuristic result in consistently lower frequency of misses and also show less variability.

mappings produced randomly and by the Hierarchical Heuristic. The median is included in the results since it can mitigate the effects of extreme sample values which can bias the mean for the small number of samples (4) used in these statistics.

Level Two Cache $MPI \times 1000$ Confidence Intervals (Mult2.2)						
Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.4953 ± 0.1264	1.4873	1.1858 ± 0.0205	1.1884	20.7%
	2	1.0480 ± 0.0186	1.0409	0.9805 ± 0.0049	0.9824	6.4%
	4	0.9593 ± 0.0038	0.9589	0.9289 ± 0.0014	0.9291	3.2%
4M	1	0.7121 ± 0.0428	0.7104	0.5982 ± 0.0062	0.5976	16.0%
	2	0.5375 ± 0.0091	0.5355	0.5101 ± 0.0027	0.5106	5.1%
	4	0.5079 ± 0.0033	0.5088	0.4938 ± 0.0009	0.4940	2.8%
16M	1	0.3339 ± 0.0235	0.3254	0.2820 ± 0.0069	0.2824	15.6%
	2	0.2539 ± 0.0034	0.2542	0.2326 ± 0.0021	0.2333	8.4%
	4	0.2371 ± 0.0041	0.2369	0.2226 ± 0.0006	0.2225	6.1%

Table 1. Mult2.2 Secondary Cache Mapping Comparison.

This table compares simulation results of the Mult2.2 trace for random virtual to real mappings and mappings produced by the Hierarchical Heuristic. Statistics are gathered for the number of misses per 1000 instructions (*MPI*). Four samples were used to produce the statistics above. The confidence intervals (mean ± error) are given together with the median of the four samples. The "Hierarchical Reduction" column indicates the decrease in the mean of the samples produced by the Hierarchical Heuristic from the mean produced by the random mappings.

The results in Table 1 show that the confidence intervals for the results obtained from the mappings of the Hierarchical Heuristic are tighter than those for the random mappings. Since the width of the confidence intervals are directly proportional to the standard deviation (because each set of statistics came from 4 samples), this indicates the smaller variance of the cache performance with careful mapping.

Table 1 also shows that the confidence intervals tend to tighten as the associativity of the secondary cache is increased. Associativity lessens the impact of the page mapping on cache performance, partly due to the fewer number of page conflicts (as shown in Figure 4), and partly due to the smaller contribution of each conflict to the frequency of misses. Appendix A presents results for the traces other than Mult2.2 and the same cache configurations as in Table 1. For brevity, confidence intervals will not be shown in the rest of the text of this paper. Mean values will be used.

The Hierarchical Reduction for direct-mapped caches is expanded in Table 2. This is the fraction of the (mean) cache misses resulting from random mapping which were eliminated by the careful mapping of the Hierarchical Heuristic.

Hierarchical Reduction for Direct-Mapped Caches			
Trace	Cache Size (Megabytes)		
	1	4	16
Mult1	13%	10%	10%
Mult1.2	7%	3%	30%
Mult2	19%	14%	13%
Mult2.2	21%	16%	16%
Tv	3%	2%	3%
Sor	1%	12%	34%
Tree	37%	25%	30%
Lin	-1%	33%	55%

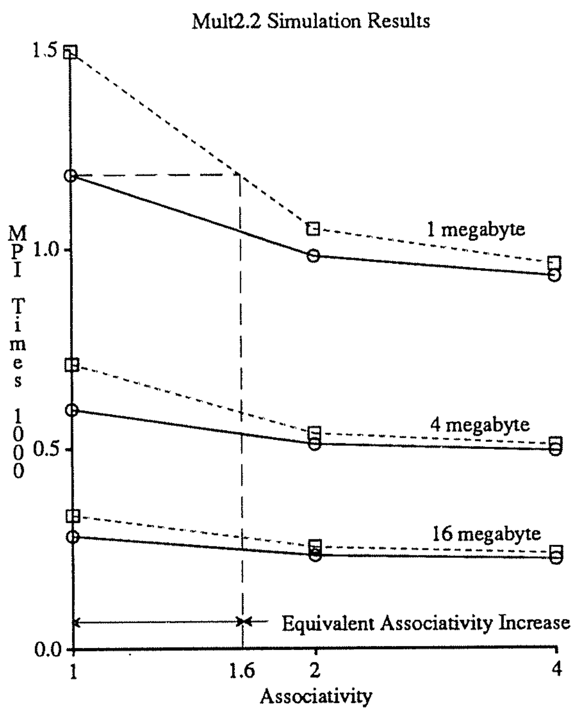
Table 2. Careful Mapping Miss Reduction for All Traces.

This figure shows, for different direct-mapped cache configurations and traces, the Hierarchical Reduction (in percent) which is the fraction of the misses produced from random mapping which were eliminated by the careful mapping of the Hierarchical Heuristic.

Comparing the (mean) frequency of misses shown in Table 2, the careful mappings reduced misses by up to 55%, while an average reduction 10-20% is possible in the direct-mapped caches. These results validate the usefulness of careful page mapping for a variety of workloads. Though there is wide variability among the results for the different traces in Table 2, and in one instance the frequency of misses slightly increased, results reasonably similar to those from the Mult2.2 trace are obtained. The Mult2.2 trace is examined in detail through the rest of this study due to its ''normal'' behavior.

The mean *MPI* values for the Mult2.2 trace from Table 1 are shown graphically in Figure 11. Even though the relative reduction to *MPI* is relatively constant and independent of cache size, the absolute value of the improvement due to careful mapping decreases with increasing cache size, since *MPI* decreases with cache size. The figure also shows that the largest gain from careful mapping comes when a direct-mapped cache is used, caches of higher associativity are affected less by the virtual to real page mapping.

By interpolating between the simulation results for different cache sizes and associativities, the effect of careful mapping relative to the effect of other cache design changes can be estimated. The dashed line in the left graph in Figure 11 attempts to determine the equivalent associativity change of Hierarchical Heuristic mapping by comparing its miss reduction on the 1 megabyte direct-mapped cache to that of changing the cache organization to 2-way. The comparison shows that careful mapping with the Hierarchical Heuristic eliminates a substantial portion of the misses of the associativity increase, making a cache perform like one of higher associativity (equivalent to a fictional associativity of 1.6). The dashed line in the right graph compares the miss reduction of careful mapping to the reduction from a cache size increase. Since careful mapping also eliminates a substantial portion of



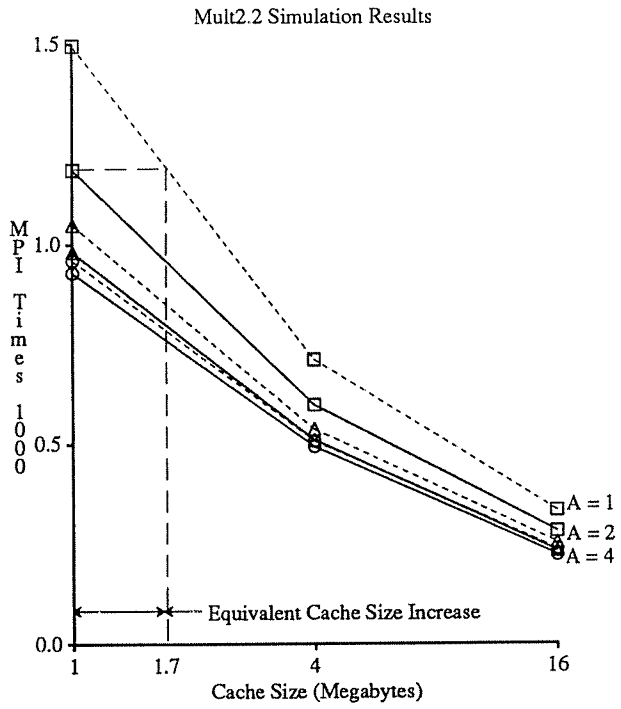


Figure 11. Mult2.2 Careful Mapping Improvement.

This figure shows graphically the (mean) Mult2.2 simulation data of the top table in Table 1. The misses per 1000 instructions for cache sizes plotted versus the cache associativity and size for random (dotted) page mappings and Hierarchical Heuristic (solid) page mappings.

The dashed lines show that Hierarchical Heuristic gets a substantial portion of the gain of increasing associativity from direct-mapped to 2-way or increasing the cache size from 1 megabyte to 4 megabytes.

the misses of a cache size increase (equivalent to a 1.7 megabyte cache), it can make a smaller cache perform like a larger cache.

4.2.2. Page Conflict Minimization

To more fully understand the causes of the miss frequency improvement due to careful mapping, Figure 12 shows the differences in the number of page conflicts (within single address spaces) produced randomly and by the Hierarchical Heuristic for the Mult2.2 trace. Each point indicates the mapping conflicts ($C - C_{\text{min}}$) from the address space of one of the processes in the Mult2.2 trace for a 1 megabyte direct-mapped cache. The Mult2.2 trace contains the execution of hundreds of processes (since many of them completed), so many sample values were available. The squares, the sampled values produced by a random mapping, follow very closely the predicted outline shown in Figure 4, validating the accuracy of the simple conflict model.

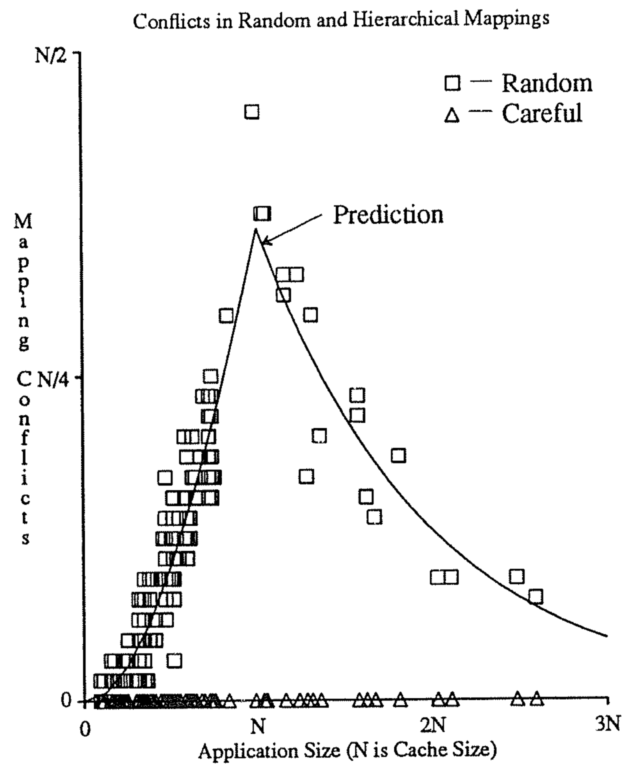


Figure 12. Conflicts for Mult2.2.

This figure shows the number of page conflicts (16 kilobyte pages) in a 1 Megabyte direct-mapped cache ($N = 64$, 128 megabyte main memory) for various processes from the Mult2.2 trace for a random mapping (squares) and a mapping using the Hierarchical Heuristic (triangles). The results show that the Hierarchical Heuristic eliminated all conflicts within the address spaces of all processes while the random mappings perform very much like that predicted in Figure 4.

The sampled values clearly show that the Hierarchical Heuristic was successful in reducing the number of conflicts in each mapping below that from random mappings. In fact, not a single address space had an unnecessary conflict from the Hierarchical Heuristic mapping, as shown in Figure 12. This is largely due to the number of pages available for mapping when each page is mapped (from the free page frame pool), four megabytes worth. On the average, there were four pages available for mapping in each bin of the one megabyte cache. These extra pages allowed the Hierarchical Heuristic sufficient freedom to obtain perfect conflict reduction in this cache.

Figure 13 establishes the relationship between the number of conflicts within an address space and the frequency of misses caused by the process using the address space (with random mapping). The *MPI* of processes referencing an amount of memory ranging from 0.5 to 1.0 megabytes are plotted versus the number of conflicts in the address space. Processes were chosen with their memory sizes within a range to reduce the effect of changes in memory size on *MPI* and isolate the effect of the number of conflicts for a given size. The particular range from 0.5 to 1.0 megabytes was chosen since there were a large number (58) of processes in this range and it is an area where substantial reductions in the number of conflicts can occur, as indicated in Figure 12 and Figure 4. The *MPI* for a process was calculated using the number of cache misses during the execution of a process. Note that this assignment of *MPI* to a process includes cache interference from within the the process itself as well as interference from other processes.

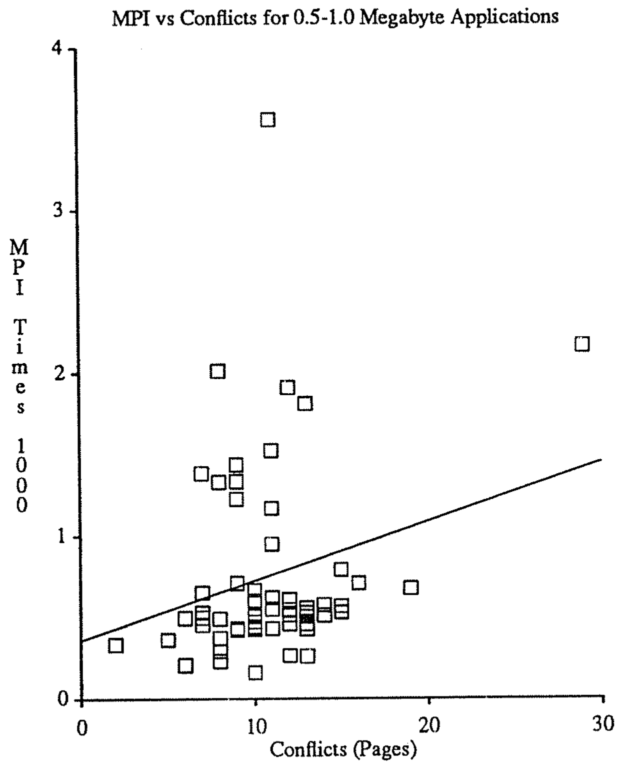


Figure 13. MPI vs. Conflicts for a Mult2.2 Random Mapping.

This figure shows the relationship between *MPI* and the number of conflicts *C* (in a 1 megabyte direct-mapped cache, 16 kilobyte pages) for the processes of size 0.5 to 1.0 Megabytes from the Mult2.2 trace with a random mapping. The solid line is the least-squares best-fit of the sample values. Though the variability is quite large, the data shows a positive correlation between the number of page conflicts and *MPI*.

The solid line in Figure 13 is the least-squares best-fit line through the sampled values. Though the variations in the relationship between *MPI* and the number of conflicts are very large, there is a slight trend toward a higher frequency of misses as the number of conflicts is increased, as evidenced by the positive slope of the best-fit line.

Together, Figure 12 and Figure 13 support the key hypothesis of this study, that techniques could be developed to minimize the number of page conflicts in a real-indexed cache resulting in significant reductions in the frequency of real-indexed cache misses.

4.2.3. Careful Mapping Alternatives

The goodness of the mappings produced by the careful mapping alternatives outlined in Section 3 depends on the size of the pool of available page frames. In the previous results for Hierarchical Heuristic maps, the pool of available page frames has been 4 megabytes. As an experiment to determine the effect of the size of the page pools on the goodness of the mappings produced by the different heuristics, Table 3 compares results for trace Mult2.2 with a 4 megabyte pool of available pages (Large Pool) and a 256 kilobyte pool (Small Pool). Shown are the *MPI* results for a 4 megabyte direct-mapped cache, which means that on the average there was 1 available page frame in each bin with the 4 megabyte pool, much less with the 256 kilobyte pool.

Level Two Cache *MPI*×1000 (Mult2.2)

Heuristic	Large Pool	Small Pool
Random	0.71 (19%)	0.71 (8%)
Page Coloring (equal)	0.69 (15%)	0.85 (30%)
Page Coloring (hash)	0.64 (8%)	0.71 (7%)
Bin Hopping	0.60 (0%)	0.67 (1%)
Sequential	0.60 (0%)	0.61 (-8%)
Hierarchical	0.60 (0%)	0.66 (0%)

Table 3. Careful Mapping Comparison.

This table compares results of the different mapping heuristics outlined in Section 3 for different page frame pool sizes for the Mult2.2 trace. Results are shown for 4 megabytes of available page frames (Large Pool) and 256 kilobytes of available page frames (Small Pool) and are given relative to the Hierarchical Heuristic for each pool size. The results shown are for a 4 megabyte direct-mapped cache.

The equality Page Coloring Heuristic performs very poorly in the results presented in Table 3 for the Mult2.2 trace since commonly used virtual addresses map to the same area in the cache. MIPS does not use strict equality match Page Coloring simulated here, likely because of this problem. The Page Coloring implementation with PID hashed matching performs better since it eliminates the problem, yet still performs poorly relative to the other careful mapping schemes. The reason for this poor performance is the limited flexibility in the Page Coloring: if an available page frame is not found in the matching bin, the search is abandoned and any arbitrary page frame is chosen. The other careful mapping schemes are more adaptable to situations with limited numbers of available page frames.

With the 4 megabyte pool of available page frames in Table 3, the Bin Hopping, Sequential, and Hierarchical Heuristics performed nearly equivalently. Each of them found it easy to eliminate conflicts when there averaged 1 available page frame in each bin. The frequency of cache misses for the mappings produced by the Sequential Heuristic is relatively independent of the page frame pool size down to 256 kilobytes (a very small amount of memory compared to the 128 megabytes of usable memory), showing the Sequential Heuristic to produce superior maps overall. The other heuristic mapping functions are negatively affected by the reduction in page pool size. The Sequential Heuristic produces the best maps, whether there is a small or large pool of available page frames.

A free page frame pool with approximately one available page per bin seems to be equivalently sufficient for conflict minimization by any of the Bin Hopping, Sequential, or Hierarchical Heuristics. This may be a good target size for the free page frame pool: *maintain a number of available page frames equal to the number of cache bins.*

Table 4 compares the performance of the different careful mappings on the Mult2.2 trace for a variety of cache configurations. The pool of available page frames is 4 megabytes. The Page Coloring, Bin Hopping, and Sequential heuristics produced page mappings which were optimized only for the 4 megabyte direct-mapped cache (or the 16 megabyte 4-way set-associative cache), shown in **Bold**. The Hierarchical Heuristic mappings were optimized for all configurations, making some of the entries in Table 4 an ``unfair'' comparison. The Hierarchical Heuristic performed substantially better than the other careful mapping schemes for the other (non-optimized) configurations, indicating the usefulness of its size independent mappings.

5. Discussion

It is important for an implementation of careful mapping to meet all the goals outlined in Section 3.1. The trace-driven simulation section has shown that implementations of careful mapping can reduce the frequency of cache misses and the variability of the cache misses for different runs. That is the most essential requirement of a careful mapping scheme.

Another important goal of a careful mapping function is that it should be low overhead, both in terms of execution time and the space required by the function. The Page Coloring and Bin Hopping Heuristics have the lowest overhead. The Sequential and Hierarchical Heuristics have more overhead, both space and time. The Sequential Heuristic could be particularly costly when the number of cache bins is large.

Level Two Cache *MPI*×1000 (Mult2.2)						
Config Size	A	Page Coloring (equality)	Page Coloring (PID hash)	Bin Hopping	Sequential	Hierarchical
1M	1	1.37 (15%)	1.32 (11%)	1.33 (12%)	1.38 (16%)	1.19 (0%)
	2	1.03 (5%)	1.02 (4%)	1.00 (2%)	1.03 (5%)	0.98 (0%)
	4	0.94 (2%)	0.94 (2%)	0.94 (1%)	0.95 (2%)	0.93 (0%)
4M	**1**	**0.69 (15%)**	**0.64 (8%)**	**0.60 (0%)**	**0.60 (0%)**	**0.60 (0%)**
	2	0.53 (3%)	0.53 (3%)	0.52 (1%)	0.52 (2%)	0.51 (0%)
	4	0.50 (2%)	0.50 (2%)	0.50 (0%)	0.50 (1%)	0.49 (0%)
16M	1	0.31 (12%)	0.31 (10%)	0.30 (6%)	0.29 (4%)	0.28 (0%)
	2	0.25 (9%)	0.25 (7%)	0.24 (4%)	0.24 (2%)	0.23 (0%)
	4	**0.24 (6%)**	**0.23 (4%)**	**0.23 (2%)**	**0.22 (0%)**	**0.22 (0%)**

Table 4. Performance of Mapping Schemes for Mult2.2.

This table compares the cache performance of mappings produced by the careful mapping heuristics for the Mult2.2 trace. The *MPI* resulting from each scheme is given relative the Hierarchical Heuristic for each configuration. The Page Coloring, Bin Hopping, and Sequential page mappings were optimized for the 4 megabyte direct-mapped configuration (and the 16 megabyte 4-way associative configuration). These configurations are shown in **bold**.

The careful mapping implementations should adapt to the features of virtual memory systems, including sharing among different virtual address spaces. The Bin Hopping Heuristic may not adapt well to sharing of pages. The problem is that each mapping function will attempt to map shared pages to bins according to the local copy of the bin pointer. Since a bin pointer is maintained per address space, each address space would map a page according to the value of its bin pointer, which is not necessarily related to the bin pointer of the other address spaces. Thus, the mapping is optimized for one address space and may cause conflicts in another.

The strict equality match implementation of the Page Coloring Heuristic adapts well to sharing when the shared pages are mapped to the same virtual address in all address spaces. In this case, the preferred bin for a virtual page is the same for each address space. More sophisticated versions of the Page Coloring Heuristic, such as the PID hash implementation considered in this study, may not adapt well to sharing for the same reason that Bin Hopping will not: the bin to which the virtual page is preferably mapped depends on the PID used to map the page. A Page Coloring implementation which, instead, maintains an offset for each sharable unit (a collection of pages, such as a shared code segment), could be better adaptable to sharing. All address spaces would then attempt to map pages similarly since the same offset is used by each address space for each sharable unit.

With the Sequential and Hierarchical Heuristic implementations of careful mapping, the *used* state information may be maintained per sharable unit to better adapt to sharing. The *used* state of an address space can then be calculated as the sum of the *used* states of each sharable unit referenced within the address space. Mapping decisions can be based on this accumulated *used* information. Otherwise, if state information is maintained only per address space, the *used* state of all address spaces accessing a page should be updated when the page is added or removed, which may be both difficult and time consuming. When the used information is maintained per sharable unit, mapping decisions for shared pages will still only be optimized with respect to the address space for which the page was mapped, but, all subsequent mapping decisions (for all address spaces referencing the page) can adjust to the positioning of the page through the updated *used* state. Note that the summation of *used* state for multiple sharable units can make the mapping process more time consuming, proving the efficiency of the Hierarchical Heuristic even more essential for good performance.

The last goal of a mapping function was to produce size independent mappings. The Hierarchical Heuristic is the only careful mapping function which achieves this goal.

6. Conclusions

The need for and implementation of careful virtual to real page mapping policies has been shown in this paper. When the size of a real-indexed cache is large, the virtual to real page mapping function and the cache mapping function interact to determine the cache performance. This gives the page mapping policy the opportunity to optimize cache performance by carefully choosing the virtual to real page mapping of an address space. It is likely to be worthwhile to expend a little effort to carefully map a page since it can improve the performance of all subsequent accesses to the page.

A simple model analyzed the potential improvement of careful mappings over random mappings. This model indicated that mapping conflicts in real-indexed caches were maximized when the size of the address space is approximately the size of the cache. In a direct-mapped cache, 30% of the address space can be unnecessarily in conflict in the cache at this point. The model also predicted that the cache is very small or very large compared to the address space size, careful mapping will be less useful and necessary.

Several heuristics were discussed to implement a careful mapping policy. These ranged from the simple Page Coloring and Bin Hopping Heuristics to the more sophisticated Sequential and Hierarchical Heuristics. The Bin Hopping, Sequential, and Hierarchical heuristics were shown to perform well under the conditions studied, with the Sequential Heuristic shown to produce the best mappings, particularly when the pool of available pages was small. The equality match Page Coloring Heuristic performed poorly with a multiprogramming workload since frequently used areas of the virtual memory space were mapped to the same cache bin. A more sophisticated version of Page Coloring that overcame this common mapping problem still did not perform as well as the other careful mapping schemes. The Hierarchical Heuristic has a number of desirable properties, especially its good computational efficiency and cache size independent mappings.

These careful mapping schemes can be very useful to reduce the frequency of misses in large real-indexed caches. Simulation results from a number of workloads were presented in this paper. The improvement of careful mapping is very dependent on the workload, but, 10-20% of direct-mapped cache misses can be eliminated by careful mapping. This is a substantial portion of the reduction achievable by doubling either the associativity or cache size. Careful virtual to real page mapping can be a very useful technique to improve the performance of large, real-indexed CPU caches.

7. Acknowledgements

We would like to thank The Western Research Laboratory of Digital Equipment Corporation, especially Anita Borg and David Wall, for the traces used in this study. Joel Bartlett, Renato De Leone, Jeremy Dion, Norm Jouppi, Bob Mayo, and Don Stark all were a tremendous help in providing traceable applications. Paul Vixie and Colleen Hawk were a great help in storing the traces. Jim Goodman, Viranjit Madan, Guri Sohi, and David Wood from the Wisconsin Architecture Group provided very valuable feedback. Paul Beebe and the Systems Lab were able to satisfy our *enormous* computing needs. Mike Litzkow and Miron Livny were a great help in adapting Condor to the requirements of these simulations. Special thanks also to Harold Stone for his very useful comments.

8. Bibliography

[AGSH86] A. AGARWAL, R. L. SITES and M. HOROWITZ, "ATUM: A New Technique for Capturing Address Traces Using Microcode," *Proceedings of the 13th International Symposium on Computer Architecture*, 1986, pp. 119-127.

[BABJ81] O. BABAOGLU and W. JOY, "Converting a Swap-Based to do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the 8th Symposium on Operating System Principles*, 1981, pp. 78-86.

[BAEW88] J. BAER and W. WANG, "On the Inclusion Properties for Multi-Level Cache Hierarchies," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 73-80.

[BOKL89] A. BORG, R. E. KESSLER, G. LAZANA and D. W. WALL, "Long Address Traces from RISC Machines: Generation and Analysis," Research Report 89/14, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, September 1989.

[BOKW90] A. BORG, R. E. KESSLER and D. W. WALL, "Generation and Analysis of Very Long Address Traces," *Proceedings of the 17th Annual International Conference on Computer Architecture*, 1990, pp. 270-279.

[FERR76] D. FERRARI, "The Improvement of Program Behavior," *IEEE Computer*, November 1976, pp. 39-47.

[GOOC84] J. R. GOODMAN and M. CHIANG, "The Use of Static Column RAM as a Memory Hierarchy," *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984, pp. 167-174.

[GOOD87] J. R. GOODMAN, "Coherency For Multiprocessor Virtual Address Caches," *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1987, pp. 72-81.

[HIEL86] M. HILL, S. EGGERS, J. LARUS, G. TAYLOR, G. ADAMS, B. K. BOSE, G. GIBSON, P. HANSEN, J. KELLER, S. KONG, C. LEE, D. LEE, J. PENDLETON, S. RITCHIE, D. WOOD, B. ZORN, P. HILFINGER, D. HODGES, R. KATZ, J. OUSTERHOUT and D. PATTERSON, "Design Decisions in SPUR," *IEEE Computer*, vol. 19, no. 10, November 1986, pp. 8-22.

[HWUC89] W. W. HWU and P. P. CHANG, "Acheiving High Instruction Cache Performance with an Optimizing Compiler," *Proceedings of the 16th International Symposium on Computer Architecture*, 1989, pp. 242-251.

[KELL90] E. KELLY, Personal Communications, 1990.

[KESS90] R. E. KESSLER, Ph.D. Work in Progress, University of Wisconsin, Madison, WI, 1990.

[MCFA89] S. MCFARLING, "Program Optimization for Instruction Caches," *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1989, pp. 183-191.

[MILF77] I. MILLER and J. E. FREUND, *Probability and Statistics for Engineers*, Prentice-Hall, Englewood Cliffs, NJ, Second Edition 1977.

[PRHH89] S. PRZYBYLSKI, M. HOROWITZ and J. HENNESSY, "Characteristics of Performance-Optimal Multi-Level Cache Hierarchies," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 114-121.

[SITA88] R. L. SITES and A. AGARWAL, "Multiprocessor Cache Analysis Using ATUM," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, 1988, pp. 186-195.

[SMIT78] A. J. SMITH, "Bibliography on Paging And Related Topics," *Operating Systems Review*, vol. 12, no. 4, October 1978, pp. 39-56.

[SMIT82] A. J. SMITH, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, September 1982, pp. 473-530.

[STAM84] J. W. STAMOS, "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory," *ACM Transactions on Computer Systems*, vol. 2, no. 2, May 1984, pp. 155-180.

[TADF90] G. TAYLOR, P. DAVIES and M. FARMWALD, "The TLB Slice -- A Low-Cost High-Speed Address Translation Mechanism," *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 355-363.

[WABL89] W. WANG, J. BAER and H. M. LEVY, "Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989, pp. 140-148.

Appendix A

This appendix shows more "raw" *MPI* results like those shown in Table 1 for each of the traces considered in this study.

Level Two Cache *MPI*×1000 Confidence Intervals (Mult1)

Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.8199 ± 0.1879	1.7851	1.5827 ± 0.0258	1.5891	13.0%
	2	1.2648 ± 0.0236	1.2561	1.2093 ± 0.0089	1.2110	4.4%
	4	1.1173 ± 0.0302	1.1061	1.0746 ± 0.0101	1.0772	3.8%
4M	1	0.8083 ± 0.0401	0.7995	0.7299 ± 0.0221	0.7367	9.7%
	2	0.5757 ± 0.0087	0.5747	0.5526 ± 0.0052	0.5534	4.0%
	4	0.5289 ± 0.0033	0.5282	0.5162 ± 0.0007	0.5163	2.4%
16M	1	0.3807 ± 0.0178	0.3782	0.3442 ± 0.0063	0.3433	9.6%
	2	0.2893 ± 0.0024	0.2895	0.2730 ± 0.0020	0.2725	5.6%
	4	0.2716 ± 0.0021	0.2718	0.2619 ± 0.0008	0.2620	3.5%

Level Two Cache *MPI*×1000 Confidence Intervals (Mult1.2)

Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.7477 ± 0.1878	1.6843	1.6305 ± 0.3856	1.4797	6.7%
	2	1.2540 ± 0.0106	1.2521	1.2035 ± 0.0108	1.2002	4.0%
	4	1.1343 ± 0.0091	1.1362	1.1102 ± 0.0061	1.1103	2.1%
4M	1	0.8880 ± 0.1837	0.8471	0.8596 ± 0.3898	0.6952	3.2%
	2	0.5873 ± 0.0082	0.5863	0.5615 ± 0.0023	0.5617	4.4%
	4	0.5405 ± 0.0026	0.5403	0.5302 ± 0.0014	0.5303	1.9%
16M	1	0.4753 ± 0.1925	0.4043	0.3343 ± 0.0173	0.3362	29.7%
	2	0.2966 ± 0.0033	0.2954	0.2788 ± 0.0051	0.2797	6.0%
	4	0.2797 ± 0.0015	0.2792	0.2703 ± 0.0012	0.2705	3.4%

Level Two Cache *MPI*×1000 Confidence Intervals (Mult2)

Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
1M	1	1.5687 ± 0.1535	1.5419	1.2742 ± 0.0224	1.2707	18.8%
	2	1.0944 ± 0.0217	1.0969	1.0076 ± 0.0036	1.0077	7.9%
	4	0.9749 ± 0.0047	0.9761	0.9417 ± 0.0022	0.9412	3.4%
4M	1	0.7360 ± 0.1067	0.6967	0.6314 ± 0.0230	0.6284	14.2%
	2	0.5462 ± 0.0056	0.5476	0.5212 ± 0.0022	0.5215	4.6%
	4	0.5152 ± 0.0021	0.5153	0.5018 ± 0.0005	0.5018	2.6%
16M	1	0.3352 ± 0.0109	0.3329	0.2931 ± 0.0047	0.2941	12.6%
	2	0.2599 ± 0.0015	0.2597	0.2392 ± 0.0010	0.2390	8.0%
	4	0.2427 ± 0.0022	0.2423	0.2294 ± 0.0003	0.2293	5.5%

Level Two Cache $MPI \times 1000$ Confidence Intervals (Tv)

Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
	1	3.3632 ± 0.1861	3.3093	3.2713 ± 0.1084	3.2610	2.7%
1M	2	2.4658 ± 0.0214	2.4633	2.4919 ± 0.0536	2.4900	-1.1%
	4	2.3055 ± 0.0094	2.3017	2.2997 ± 0.0052	2.3006	0.2%
	1	2.1151 ± 0.0549	2.1263	2.0682 ± 0.0268	2.0684	2.2%
4M	2	1.8294 ± 0.0079	1.8271	1.8108 ± 0.0067	1.8103	1.0%
	4	1.7800 ± 0.0037	1.7802	1.7772 ± 0.0045	1.7764	0.2%
	1	1.1899 ± 0.0062	1.1888	1.1589 ± 0.0293	1.1513	2.6%
16M	2	1.0337 ± 0.0043	1.0337	1.0171 ± 0.0051	1.0170	1.6%
	4	0.9910 ± 0.0027	0.9910	0.9779 ± 0.0031	0.9768	1.3%

Level Two Cache $MPI \times 1000$ Confidence Intervals (Sor)

Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
	1	14.9794 ± 0.0211	14.9757	14.8567 ± 0.0284	14.8627	0.8%
1M	2	14.7033 ± 0.0299	14.6937	14.6402 ± 0.0053	14.6401	0.4%
	4	14.5816 ± 0.0152	14.5774	14.5507 ± 0.0040	14.5496	0.2%
	1	9.4718 ± 0.0983	9.4729	8.3389 ± 0.1115	8.3545	12.0%
4M	2	8.6826 ± 0.0581	8.6821	8.0557 ± 0.1145	8.0312	7.2%
	4	8.3524 ± 0.0291	8.3509	8.0611 ± 0.0323	8.0529	3.5%
	1	4.2208 ± 0.3183	4.1976	2.7843 ± 0.1638	2.7574	34.0%
16M	2	2.8435 ± 0.1069	2.8471	2.1481 ± 0.0824	2.1383	24.5%
	4	2.4179 ± 0.1447	2.3788	2.0743 ± 0.0197	2.0754	14.2%

Level Two Cache $MPI \times 1000$ Confidence Intervals (Tree)

Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
	1	3.9061 ± 2.0853	3.3053	2.4720 ± 0.0573	2.4855	36.7%
1M	2	2.0930 ± 0.0580	2.1085	1.9738 ± 0.0516	1.9561	5.7%
	4	1.8580 ± 0.0494	1.8476	1.8573 ± 0.0584	1.8392	0.0%
	1	1.2112 ± 0.2466	1.1478	0.9124 ± 0.0286	0.9174	24.7%
4M	2	0.7179 ± 0.0292	0.7124	0.5934 ± 0.0284	0.5848	17.3%
	4	0.5292 ± 0.0218	0.5260	0.4932 ± 0.0119	0.4897	6.8%
	1	0.5416 ± 0.2284	0.4682	0.3771 ± 0.0292	0.3805	30.4%
16M	2	0.2914 ± 0.0194	0.2859	0.2706 ± 0.0047	0.2707	7.1%
	4	0.2579 ± 0.0008	0.2578	0.2545 ± 0.0002	0.2544	1.3%

Level Two Cache $MPI \times 1000$ Confidence Intervals (Lin)

Configuration		Random		Hierarchical		Hierarchical
Size	Assoc.	90% Confidence	Median	90% Confidence	Median	Reduction
	1	1.1481 ± 0.0107	1.1511	1.1609 ± 0.0021	1.1616	-1.1%
1M	2	1.0806 ± 0.0037	1.0800	1.0938 ± 0.0010	1.0935	-1.2%
	4	1.0611 ± 0.0030	1.0613	1.0766 ± 0.0005	1.0766	-1.5%
	1	0.5277 ± 0.0263	0.5264	0.3564 ± 0.0283	0.3488	32.5%
4M	2	0.3103 ± 0.0392	0.3054	0.1307 ± 0.0234	0.1230	57.9%
	4	0.1557 ± 0.0275	0.1567	0.0494 ± 0.0108	0.0452	68.2%
	1	0.1532 ± 0.0230	0.1529	0.0689 ± 0.0063	0.0692	55.0%
16M	2	0.0416 ± 0.0101	0.0438	0.0217 ± 0.0038	0.0204	47.8%
	4	0.0179 ± 0.0006	0.0178	0.0167 ± 0.0000	0.0167	6.7%