Page Placement Algorithms for Large Real-Indexed Caches[†]

R. E. Kessler
Mark D. Hill

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

ABSTRACT

When a computer system supports both paged virtual memory and large real-indexed caches, cache performance depends in part on the main memory page placement. To date, most operating systems place pages by selecting an arbitrary page frame from a pool of page frames that have been made available by the page replacement algorithm. We give a simple model that shows that this naive (arbitrary) page placement leads to up to 30% unnecessary cache conflicts. We develop several page placement algorithms, called *careful-mapping algorithms*, that try to select a page frame (from the pool of available page frames) that is likely to reduce cache contention. Using trace-driven simulation, we find that careful-mapping results in 10-20% fewer (dynamic) cache misses than naive mapping (for a direct-mapped real-indexed multi-megabyte cache). Thus, our results suggest that careful-mapping by the operating system can get about half the cache miss reduction that a cache size (or associativity) doubling can.

1. Introduction

Most general purpose computer systems support both paged virtual memory [DENN68] and caches [SMIT82]. Virtual memory caches pages by translating virtual addresses to their corresponding real addresses (often by consulting a page table). This translation from virtual pages to real page frames is usually fully associative: *any* page frame in the physical main memory can hold any page. In contrast, hardware caches usually have a more restrictive set-associative mapping that only allows a cache block to reside in one of the few frames in a set. A real-indexed (virtual-indexed) selects the set for an incoming reference by extracting index bits from the

† A preliminary version of the material presented here appeared as a technical report [KESH90]. This 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, Cray Research, and Digital Equipment Corporation. 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. He currently is employed at Cray Research, Inc., Chippewa Falls, WI.

reference's real (virtual) address.

Figure 1 depicts virtual address translation together with cache set-indexing. A large real-indexed cache requires so many index bits that some of them must come from the address translation output[1]. We call the index bits coming from the address translation the *bin-index* and the group of sets selected by them a *bin*[2]. Address translation affects large real-indexed cache performance because page translation (implicitly) selects the cache bin that the cache blocks in the page reside in. In this study, we examine this interaction between page placement and cache performance. We focus on large real-indexed caches because page placement has no affect on a cache that is either small or virtual-indexed.

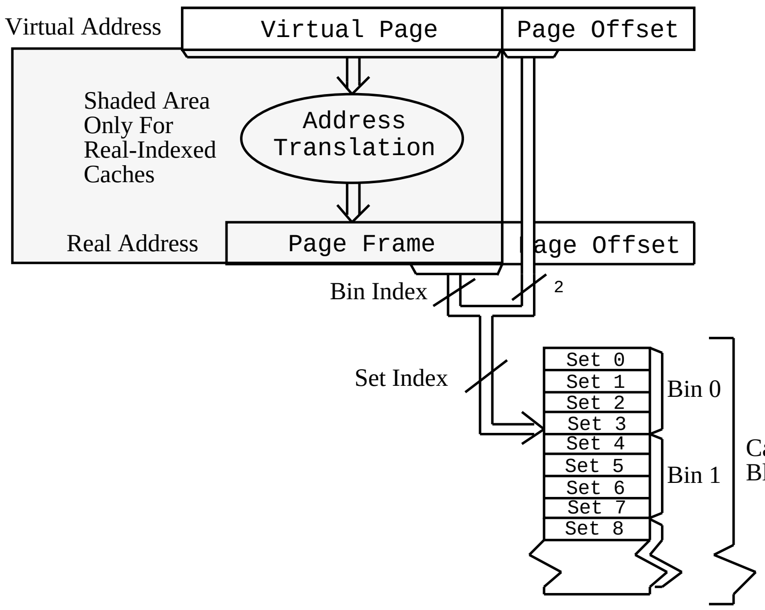

Figure 1. Virtual Memory and Set-Indexing Interaction.

This figure shows translation stages that index a virtually addressed memory location into a large set-associative CPU cache. The cache takes index bits directly from the virtual address for a virtually-indexed cache, and it takes the bits from the real address for a real-indexed cache. Any of the *A* (*A* is the associativity) cache block frames in the set chosen by the *index* bits can contain the addressed memory location. This figure depicts bins that are four cache sets. This paper only considers large real-indexed caches with bit-selection indexing [SMIT82].

Two trends warrant an examination of the interaction between virtual memory page mapping and large real-indexed caches. First, real-indexed caches are commonly implemented because they handle virtual address synonyms (aliases) and virtual address space changes (e.g., context switches) more simply than do virtual-indexed

1. A large cache is defined to be one whose size divided by its associativity exceeds the page size. The associativity of the cache is the number of cache block frames in each set.

2. Note that the *superset* of Goodman [GOOD87] is different from a bin. Pieces of a superset span *all* bins.

caches [SMIT82]. Real-indexing is particularly appropriate for secondary caches (e.g., level-two caches: those that service the references from primary, or level-one, caches) in hierarchical CPU cache configurations, because the address translation cost of real-indexing has a smaller performance impact [TADF90, WABL89]. Second, typical cache sizes are growing, so large caches, even multi-megabyte ones, are now feasible design alternatives [BOKW90]. Page mapping has a greater effect on these large caches because they have many more bins.

1.1. Previous Work

Few previous studies examine the page mapping interaction with real-indexed cache performance because large real-indexed caches are a recent trend. Sites and Agarwal [SITA88] compare the performance of virtual-indexed caches to real-indexed caches. They find that real-indexed caches perform worse than their corresponding virtual-indexed caches unless there is frequent context switching.

Only a few systems optimize their page placements for cache performance. In the Sun 4/110, the operating system maps instruction pages and data pages [KELL90] to even and odd page frames to partition the instructions and data in the 4/110's SCRAM cache [GOOC84]. MIPS uses a variant of *Page Coloring*, described in Section 3, to improve and stabilize its real-indexed cache performance [TADF90]. With trace-driven simulation, Bray, et al., [BRLF90] verify that Page Coloring reduces the mean and variance of the cache miss ratio, particularly for large caches. They also model the effective main memory size reduction of a strict version of Page Coloring that assumes a page *must always* be placed in a frame of the correct color. This model does not apply to our work (or to MIPS' implementation) because it implies that page replacement may be required even when frames (of the wrong color) are available. Rather, we require page replacement *only* when there are no available page frames.

Finally, many others [FERR76, HWUC89, LARW91, MCFA89, STAM84] have examined changing virtual-address reference patterns to improve cache performance. These techniques can be used in conjunction with our work, since our techniques alter real addresses, not virtual addresses.

1.2. Contributions of this Paper

Operating systems place (map) new pages into main memory by selecting from a pool of available page frames. This pool consists of page frames that have not been used recently and can be reallocated to new pages. The selection process is usually arbitrary; often the first available frame in the pool is reclaimed by the new page. We call this *random* (or naive) page mapping because the operating system does not care *where* it places the new page. We introduce better page placement algorithms that we call *careful page mapping* algorithms. They are small modifications to the existing operating system virtual memory management software that improve *dynamic* cache performance with better *static* page bin placement decisions. When the pool of available page frames allows some mapping flexibility, rather than just choosing an arbitrary page (like the first available), a careful mapping algorithm chooses one of the available page frames that best suits some simple static cache bin placement heuristics.

These careful mapping algorithms are low-overhead, particularly since they may only execute once when each page is mapped, while they may improve cache performance on each reference to the page. Furthermore, our careful mapping algorithms need not impact the page fault rate by changing page replacement decisions, because they never force a page replacement when there are any page frames in the available page frame pool, and because the contents of the available page frame pool are determined solely by the operating system's page replacement algorithm, not by the careful mapping algorithms. Our algorithms provide the opportunity for cache performance improvements when the operating system has more frames available for placement. When main memory is under-utilized, careful mapping algorithms are a cheap way to improve cache performance because it is easy to have a large available pool. When main memory is more highly-utilized, only a small pool will be available so careful mapping will be less effective (and less needed because page swapping may dominate performance).

Section 2 motivates the static improvements that these careful mapping algorithms rely on. It first shows how the operating system can improve static page bin placement. A simple static analysis then shows the potential careful mapping static improvements: as many as 30% of the pages from an address space unnecessarily conflict in a direct-mapped cache when using random mapping. This analysis also correctly predicts that the largest gain from careful page mapping comes when the cache is direct-mapped.

Section 3 describes Page Coloring and introduces several more careful page mapping algorithms. *Page Coloring* matches the real-indexed bin to its corresponding virtual-indexed bin. *Bin Hopping* places successively mapped pages in successive bins. *Best Bin* selects a page frame from the bin with the fewest previously allocated and most available page frames. *Hierarchical* is a tree-based variant of Best Bin that executes in logarithmic time, and produces cache size independent placement improvements (i.e. it improves static placement in many different cache sizes simultaneously).

Section 4 discusses some of the methods used to get the trace-driven simulation results of this paper. Section 5 uses the simulation results to measure the cache miss reduction from the careful mapping algorithms. It shows that careful page mapping reduces the direct-mapped cache miss rate by 10-20% with our traces. This is about half of the improvement available from either doubling the size or the associativity of a direct-mapped cache. Section 5 also compares the performance of the different careful page mapping implementations for one multiprogrammed trace, showing that the Page Coloring implementations perform poorly, and that Best Bin results in the fewest cache misses.

The implication of this research is that careful mapping can improve the performance of systems with large real-indexed caches. We believe that careful mapping is appropriate for future operating system implementations since it improves cache performance with small software changes and no hardware changes.

2. Motivation for Careful Page Mapping

This section shows the potential static real-indexed page placement improvement of careful mapping over random mapping. Figure 2 depicts two placements of the code, data, and stack pages from an address space in the real-indexed cache; the left one might be a random placement of the pages and the right one might be a careful placement of the same pages. The random placement puts many pages in the same cache bins; this competition is undesirable since it can cause more cache *conflict misses* [HILS89]. Some of the randomly-mapped pages could be placed in the unused bins to reduce conflict misses.

We quantitatively measure the potential static careful mapping improvement by counting the *page conflicts*, C, which is the number of pages in a bin beyond the associativity of the cache. If u pages land in a cache bin with associativity A, then we say $max(0, u - A)$ conflicts occur[3]. With a direct-mapped cache there are six conflicts in the random mapping of Figure 2, and there are four conflicts in the careful mapping. Similarly, with a two-way set-associative cache, there are three and zero conflicts.

We can calculate the average page conflicts from random mapping if we assume each page frame is independently and equally likely to hold a page[4]. Let N be the cache size in pages and $B = N/A$ be the number of bins. A binomial distribution gives the probability that exactly u of U pages fall in one of the B bins[5]:

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

3. We use u-A rather than u because we feel that it more correctly predicts the magnitude of the conflict. If we instead used u, then conflicts would be minimized by first filling the cache (placing A pages in each bin), and then placing all the rest of the pages in a single bin; it is unlikely that this mapping would minimize dynamic cache misses.

4. We assume an infinite main memory, but the results for a finite main memory are similar for the parameters we consider.

5. Both Thiebaut and Stone [THIS87] and Agarwal, et al. [AGHH89] use binomial distributions for similar analysis.

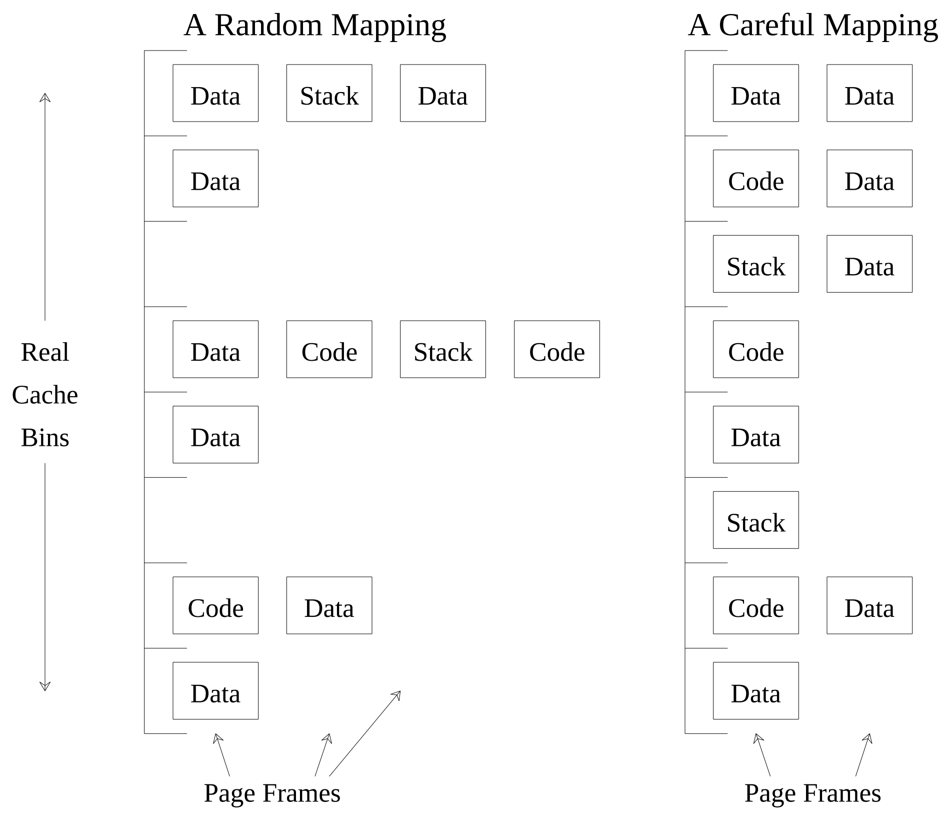

Figure 2. Random and Careful Real-Indexed Bin Placements.

This figure shows two mappings of the code, data, and stack pages of an address space into a real-indexed cache. The left bin placement might result from random mapping and the right one might be a careful mapping. The horizontally stacked pages are the (competing) ones that the virtual to real address translation indexes to the same bin.

Using Equation 1, Equation 2 calculates the expected conflicts resulting from a random placement of *U* pages across all cache bins:

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

The left graph in Figure 3 plots C_{avg} and its bounds for various address space sizes (U) with a direct-mapped cache. In the worst case, almost all the pages cause conflicts because they are placed in the same bin, so $C_{\max}$ is linear with the address space size. In the best case, the pages are evenly spread across the cache bins, so no conflicts occur until the cache is full of pages, and then each additional page adds a new conflict. C_{avg} remains closer to $C_{\min}$ than $C_{\max}$, particularly for small and large values of U.

The right graph in Figure 3 illustrates the difference between C_{avg} and $C_{\min}$ more dramatically for direct-mapped, two-way, and four-way set-associative caches. It plots what we call the *mapping conflicts* ($C_{avg} - C_{\min}$),

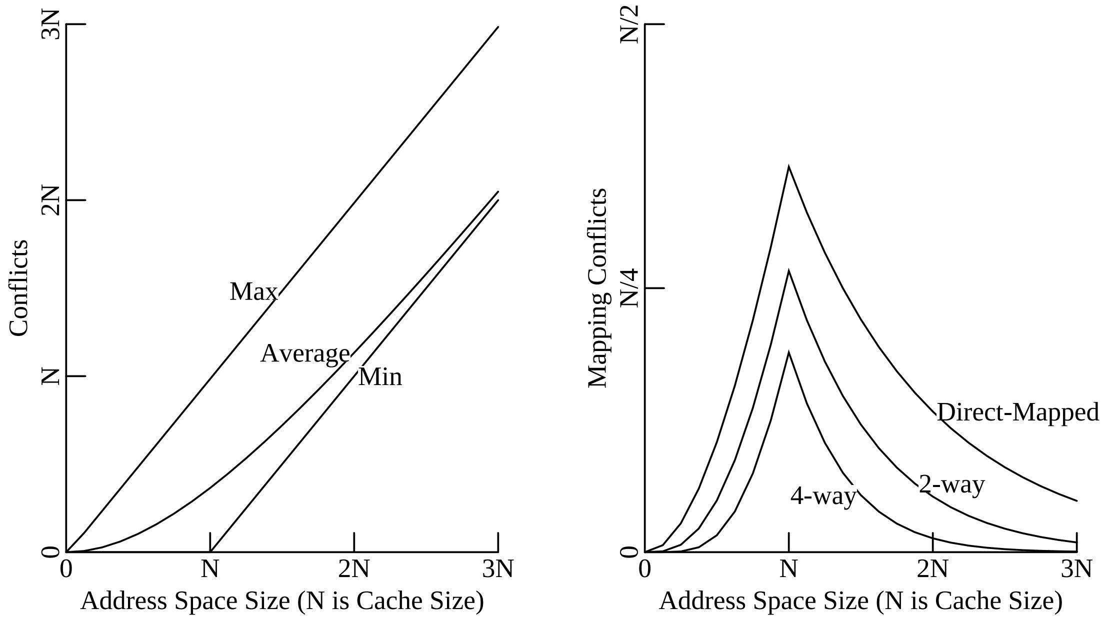

Figure 3. Static Cache Placement Conflicts.

This figure shows the potential improvement of careful mapping by comparing the average number of static page conflicts in a random mapping to bounds. The left graph shows C_{avg} (as calculated by Equation 2), $C_{\max}$ ($max(0, U-A)$), and $C_{\min}$ ($max(0, U-N)$) for various address space sizes (U) with a direct-mapped ($A=1$) 64-bin cache ($N=B=64$). The right graph shows the difference between the conflicts resulting from a naive mapping and the minimum conflicts, the mapping conflicts ($C_{avg}-C_{\min}$), for different address space sizes and cache associativities ($A=1,2,4$, $N=64$, $B=N/A$).

which is the random conflicts less the minimum, and the potential static conflict savings of careful page mapping. It shows that the mapping conflicts are maximized at the point where the address space size equals the cache size. $C_{avg}-C_{\min}$ is low for small address spaces since many collisions by few pages is unlikely. It is also low for large address spaces since the bins are likely to be highly utilized regardless of the page mapping, so there is little potential for improvement. This data shows that careful page mapping can reduce the static conflicts from random mapping; as many as 30% of the pages from an address space are unnecessarily in conflict with random mapping. It also shows that careful mapping is most useful for direct-mapped caches because they have more static page conflicts. In the opposite extreme, there are never any mapping conflicts in a fully associative cache.

3. Implementing Careful Mapping

This section presents four careful page mapping implementations. A key aspect of all these algorithms is that they do not use any dynamic referencing information to improve page placements. Rather, they place pages into the bins of a large unified[6] cache using solely static criteria. Static placement decisions are appropriate

6. We do not consider mapping functions that statically improve the page placement in split caches. An example of split caches is split instruction and data caches [SMIT82]. With split caches, the mapping function may want to minimize the contention in both caches concurrently (or independently).

because dynamic information, such as cache references (misses) or page references (faults), may not be available to the page mapper. Section 5 shows that the static improvements of these algorithms eliminate many dynamic cache misses.

We consider systems in which multiple address spaces share the same main memory, so both intra-address-space and inter-address-space conflicts occur. Except for certain obvious circumstances, the careful mapping algorithms primarily try to remove intra-address-space conflicts, and only secondarily try to reduce inter-address-space conflicts. This is appropriate for future high-performance processors because the memory within a *single* address space may be referenced many thousands, or even millions, of times before any inter-address-space conflicts cause any cache misses (only as a result of context switches).

In order to strictly differentiate between page *re*placement (making a frame available for reuse by placing it in the available pool) and page placement (choosing a frame from the pool), the careful mapping algorithms always select frames that were previously in the available pool. Thus, page replacement needs to occur only when the pool is empty[7]. Since the placement algorithms do not *require* an available pool with any specific characteristics, and many operating systems try to maintain a pool anyway (to smooth out peaks in memory demand, for example [BABJ81]), we consider placement independently from how the available pool is replenished (either by page deallocation or replacement). The placement algorithms need not force *any* page replacement modifications, though clearly the size of the available pool can determine the page placement quality, and too large of a pool can increase the page fault rate because of poorer page replacement decisions. Since the pool size may be limited, careful mapping algorithms should preferably produce better placements with smaller pools.

3.1. Page Coloring

The simplest careful mapping technique simply makes a real-indexed cache perform like a virtual-indexed cache by matching bits of the virtual page with bits from the real page frame number. MIPS has implemented a variant of this, called *page coloring* [TADF90]. In our simple implementation of Page Coloring, the page mapping function simply examines the pool of available page frames to find one with the proper bit value(s). When it finds a match, it extracts the frame and places the page in it. When there is not a matching frame in the pool, our implementation chooses an arbitrary frame (the first available in the pool). Given the complexity of current memory systems, the simplicity of Page Coloring is a virtue.

7. Bray, et al. [BRLF90], on the other hand, assume that a page is alway placed in a frame of the correct color, even if a page replacement must be initiated to get one.

Page Coloring minimizes cache contention because sequential virtual pages do not conflict with each other in the cache. Mostly-contiguous address spaces (the common case) will usually have little cache contention with Page Coloring. However, the exact match of virtual bits with real bits, as described above, can lead to excessive inter-address-space contention in a multiprogrammed workload because it places commonly-used virtual addresses from different address spaces (like the stack, for instance) in the same bin. We consider two versions of Page Coloring: (1) an exact match of bin bits, and (2) a match of bin bits exclusive-ored with the process' identifier (PID-hashing). PID-hashing makes it unlikely the same virtual address from two different address spaces maps to the same bin, thereby reducing contention when some virtual addresses are frequently used (e.g., if the top of stack for all processes is at the same virtual address). It is our understanding that MIPS uses a variant of PID-hashing [TADF90].

3.2. Bin Hopping

Bin Hopping is another simple way to equalize cache bin utilization. It allocates sequentially mapped pages in sequential cache bins, irrespective of their virtual addresses. Bin Hopping reduces cache contention because it sequentially distributes the pages from an address space across different bins (until it must wrap-around). It exploits temporal locality because the pages it maps close in time tend to be placed in different bins. If pages mapped together are also referenced together, this will be a useful property for reducing cache contention.

The Bin Hopping implementation simply traverses the bins until it finds one of them that has an available page frame. Then it remembers where it placed the last page, so the next traversal can start from there. It maintains a separate traversal path per address space to minimize contention within each address space separately. The simplicity of Bin Hopping, like Page Coloring, is its largest asset. An advantage of Bin Hopping over Page Coloring is that it is more flexible when there are few pages available. Instead of choosing an arbitrary page when one is not available in the desired bin, it skips to another bin that is almost as good as the previous one. Bin Hopping is able to do this because it remembers the last page placement for each address space[8].

3.3. Best Bin

The strength of Page Coloring and Bin Hopping could also be considered their weakness. They both have simple implementations that require little (or no) storage, but their decision-making is not very sophisticated. When there are few available page frames, Page Coloring and Bin Hopping may not produce a good mapping.

8. We initialize this state to a random location before any mapping occurs.

Best Bin uses the counter pair <*used, free*> associated with each bin to select a bin. *Used* is the number of previously placed pages from a given address space, and *free* is the number of page frames available (system-wide) in this bin[9]. Best Bin reduces contention by placing a page in the bin with the smallest *used* and largest *free* value. The choice of the smallest *used* value minimizes intra-address-space contention because all bins become equally used by each address space, and the secondary choice of the largest *free* value tends to minimize inter-address-space contention because large free values may imply bin underutilization.

As an example of a choice made by Best Bin, consider a cache with four bins and <*used, free*> pairs of <0, 0>, <1, 3>, <1, 1>, and <2, 4>. Best Bin would select the <1, 3> bin. <0, 0> could not be selected because there are no available frames, even though it has the minimal *used* count. <1, 3> is better than <1, 1> because there are more available frames, and <1, 3> is better than <2, 4> because its *used* count is smaller. After choosing it, Best Bin modifies the <1, 3> bin to <2, 2> to reflect the altered system state.

3.4. Hierarchical

The execution time of Best Bin may be linear in the number of bins, which is too slow when there are too many bins. Alternatively, Hierarchical can choose a bin in logarithmic time with a tree whose leaves contain <*used, free*> values for each bin and where each interior node value contains the sum of the values of its children[10]. Hierarchical selects a bin by beginning at the tree's root and traversing down to it leafs. At each interior node, Hierarchical chooses a traversal direction by comparing <*used, free*> values of the children, using same priorities as Best Bin.

Figure 4 shows Hierarchical on a simple example. Traversal starts from the root of the bin tree. From there, Hierarchical decides based on the <4, 6> and <5, 5> pairs corresponding to its sub-trees. It chooses <4, 6> since it has less *used* pages. Next, Hierarchical chooses <1, 3> over <3, 3>, again because there are less *used* pages. Finally, it chooses <1, 3> over <0, 0> because <1, 3> has available page frames. Bin tree traversal is then complete since a leaf node is reached. Note that Best Bin would choose the same bin in this example. To update the state to account for the page mapping, Hierarchical modifies the values of the traversed nodes to <2, 2>, <2,

9. While it is possible to determine these counts from the complete mapping of virtual pages to page frames, Best Bin can store them in a more readily-available form for greater execution efficiency. The added storage required by Best Bin is the *used* counter per bin in each address space, since the space to store the single system-wide *free* counters is minimal. Even the *used* counter storage is modest when compared to the complete page mapping information.

10. Hierarchical actually uses a single system-wide free bin tree, but it needs a used bin tree with each address space, just like Best Bin needs a bin array per address space. The storage for the used bin tree need not be more than twice that required for a bin array, still a small amount compared to the complete page mapping information. This study uses binary trees; higher branching factors also could be used, but they may not maintain size independence with all caches.

2>, <5, 5>, and <10, 10> from bottom to top, respectively.

An important feature of Hierarchical is its cache size independence. Hierarchical can simultaneously optimize page placement for many caches sizes if the tree is constructed so the children of each internal node have bin numbers with the most least-significant bits in common [KESS91]. The tree for eight bins, for example, has eight leaf nodes 000, 001, ..., 111. One level higher up in the tree, there are only four nodes: 00, 01, 10, 11. For Hierarchical to be size-independent, 00 should parent 000 and 100, 01 should parent 001 and 101, 10 should parent 010 and 110, and 11 should parent 011 and 111. That way, each tree level selects one more (high-order) bit, or equivalently, optimizes for the next larger cache. For example, size-independence is extremely important with a hierarchy of large real-indexed caches, since the performance of *all* caches (e.g. both primary and secondary) can be improved with a *single* size-independent page placement. Size-independent page placement also has a simulation advantage: many different-size caches can be simultaneously simulated.

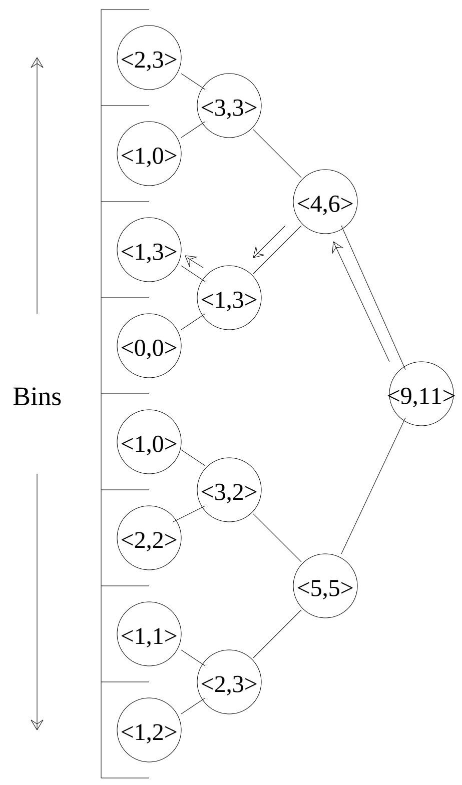

Figure 4. Bin Tree Traversal.

This figure shows an example of a bin choice when mapping a virtual page by <*used*, *free*> bin tree traversal with Hierarchical. The values of the tree nodes are given. The used values of the leaves show the number of pages already placed in the corresponding bin. The free values of the leaves show the number of page frames available for replacement in the corresponding bin.

4. Trace-Driven Simulation Methodology

This section discusses some of the methods used for the simulation results shown in Section 5.

4.1. The Traces

The traces used in the study were collected at DEC Western Research Laboratory (WRL) [BOKL89, BOKW90] on a DEC WRL Titan [NIEL86], a load/store (‘‘RISC’’) architecture. Each trace consists of the execution of three to six *billion* instructions of large workloads on a load/store architecture, including multiprogramming but not operating system references. Each trace references from 8-megabytes to over 100-megabytes of unique memory locations. The traces are sufficiently long to overcome the cold-start intervals of even the very large caches considered in this study; we present cold-start results, but the error cannot be more than a few percent for even the largest cache. We chose programs with large memory requirements because large application sizes will likely be more common as main memories of hundreds of megabytes become common.

The traces of the multiprogrammed workloads represent the actual execution interleaving of the processes on the traced system with switch intervals that may be appropriate for future high-performance processors. 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 (about 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 that uses large, sparse matrices (62-megabytes). **Tv** is a VLSI timing verifier that first builds a large data structure and then traverses it with little locality (96-megabytes). **Tree** is a Scheme program that searches a large tree data structure (64-megabytes). **Lin** is a power supply analyzer that uses sparse matrices (57-megabytes).

4.2. The Simulator

Previous trace-driven cache simulation results either assume small or virtual-indexed caches, or that operating system policies are fixed and independent of the studied caches. Our simulator incorporates memory management policies of the operating system along with hardware cache management implementations because we examine the interaction of hardware and software memory system components. Consequently, the results expose the interaction between hardware and software system components.

We arbitrarily chose a global least-recently-used (LRU) replacement policy for this study. The simulator maintains an exact ordering of the page frames from most recently used to least recently used (an LRU list, randomly initialized when simulation begins, to model a system that has been executing for some time). We assume a page placement policy can select any page frame in a pool of available page frames the end of the LRU list. (The simulator does not place frames in the available pool when a process exits and its pages are deallocated, only when they are least-recently-used.) The size of the available page frame pool is constant throughout the simulations of this study, even at the beginning of a simulation when the entire main memory is unmapped. The default pool size is 4-megabytes of 16-kilobyte pages. The main memory size of 128-megabytes is large enough that the simulations required no process swapping, so we use the static execution interleaving captured on each trace to model context-switching.

The simulated CPU cache configuration is a two-level hierarchical system that does not maintain multi-level inclusion [BAEW88]. The primary (level-one) caches are split direct-mapped write-allocate, write-back[11] instruction and data caches of 32-kilobytes each with 32-byte blocks. They are backed up by a 1-megabyte to 16-megabyte unified secondary (level-two) write-back cache with a random replacement policy and 128-byte blocks. We simulate many different secondary cache configurations, and we focus on the large secondary caches since they are most likely to be real-indexed and have the most to gain from careful page mapping.

4.3. Misses Per Instruction

Our simulation tools report cache performance in *misses per instruction* (MPI). In this study, we only compare alternatives for the large secondary caches, so the MPI results are proportional to global miss ratio (cache misses divided by processor references [PRHH89]), and relative changes in MPI are equal to relative changes in global miss ratio. We do not use local miss ratio (cache misses divided by cache accesses [PRHH89]), because careful mapping can cause the local miss ratio of a secondary cache to increase even as system performance improves (by reducing primary cache misses and hence secondary cache accesses). Kaplan and Winder [KAPW73] provide good arguments for using misses per instruction.

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

5. Trace-Driven Simulation Results

This section first shows that the static page placement improvements of careful page mapping (*Hierarchical*) eliminate many dynamic cache misses. Then it compares the alternative careful mapping implementations using one of the multiprogrammed traces.

5.1. The Usefulness of Careful Page Mapping

We compare Hierarchical mapping to random mapping in this section because Hierarchical's size-independence reduces simulation time, and because Hierarchical performs well compared to the other careful mapping schemes (as will be shown in Section 5.2). For 1-megabyte (top), 4-megabyte, and 16-megabyte caches (bottom), Figure 5 plots 100 million instruction averages of the Mult2.2 misses per instruction (*MPI*) for four different simulations, each with a different virtual to real page mapping. The cache performance is different for each simulation because they each place the same pages in different frames. Note that random mapping *does not* imply random page replacement; it means that the simulation places the page in the page frame at the bottom of the LRU list. Thus, in the random runs the simulator places the page in a random (arbitrary) bin that is determined by the page frame number at the bottom of the LRU list.

Figure 5 shows, for the Mult2.2 trace, that the careful (Hierarchical) mappings have consistently lower *MPI* than random (since the solid lines are usually below the dotted lines). This clearly indicates that the static page placement improvements of careful mapping do reduce dynamic cache misses for this trace. Over most of the trace, the careful mapping *MPI*'s are only a few percent better than random. But in addition to this small and constant improvement, there are bursts of cache misses from poor random page mapping that occasionally double or triple the *MPI* for the 100 million instruction intervals[12]. These bursts increase the mean improvement of careful mapping beyond only a few percent.

To get a better idea of the quantitative dynamic cache performance improvements of careful mapping, we wish to determine the average cache *MPI* when using random and Hierarchical mapping. To do this, we averaged the results from multiple simulation runs that use different page mappings. While the *MPI* of one simulation is an unbiased estimate of the true mean *MPI*, Figure 5 suggests that the *MPI* results from simulations can substantially vary, so we don't know how close one simulation result is to the true mean *MPI*. With a sample of size four, the

12. Note that some bursts appear in the random mappings of both the 1-megabyte and 4-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. The results for each individual cache are not correlated, however.

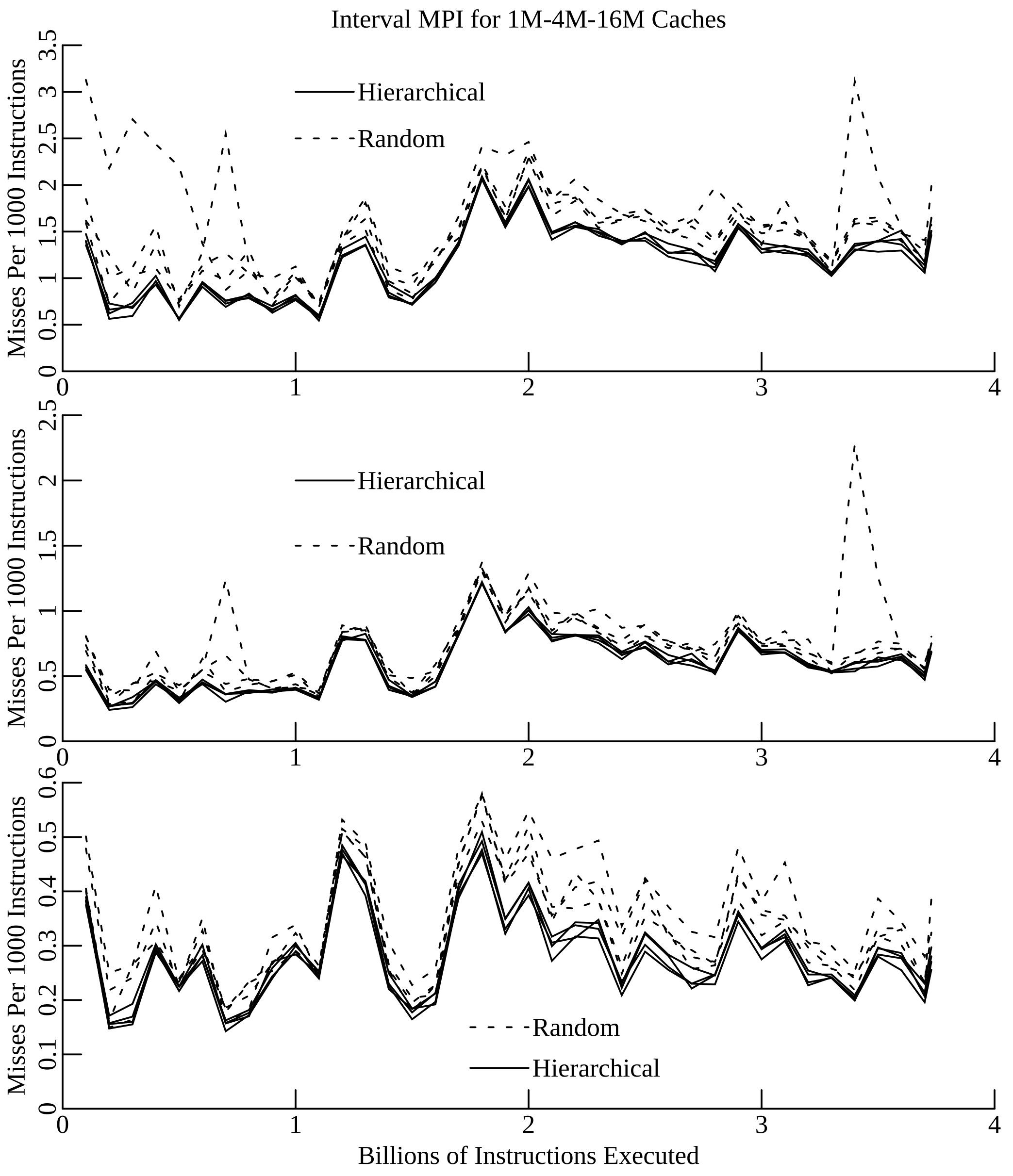

Figure 5. Mult2.2 Results for Different Page Mappings.

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

variations across the simulations were sufficiently small that the four-sample mean could serve as a good estimate of the true mean (most simulations are within a few percent of the sample mean). It seems that each trace was long enough that the individual simulation runs already averaged out enough of the variance in cache performance. Luckily, a sample of size four was sufficient, since each long-trace simulation takes days. With shorter traces, we likely would have needed many more simulations to obtain the same accuracy[13]. Below, we use four-sample means to quantitatively evaluate the improvement of careful mapping for 1-megabyte, 4-megabyte, and 16-megabyte secondary caches with direct-mapped, two-way, and four-way set-associativity.

Figure 6 shows the direct-mapped Hierarchical miss reductions. This gives the relative amount that the four-sample Hierarchical *MPI* is lower than the four-sample random mapping *MPI*. The figure shows that careful page mapping is useful for a variety of workloads. Careful mappings eliminate up to 55% of the direct-mapped cache misses from random mapping. On the average over all the traces, it reduces the random-mapping direct-mapped cache misses per instruction by about 10-20%. A similar look at the results for two- and four-way associativities suggests that careful mapping eliminates about 4-10% of the misses in a two-way set-associative cache with random replacement, and about 2-5% of the misses in a four-way set-associative cache with random replacement (on average). This verifies that the largest gain from careful mapping comes with a direct-mapped cache, as predicted by the simple static analysis of Section 2. Higher set-associativities can more easily reduce contention without static placement improvements because they spread contending blocks across the cache block frames in a set. Static improvements that avoid contention are more valuable with direct-mapping because direct-mapped caches cannot handle contention for the same set (block frame).

The direct-mapped Hierarchical miss reduction is positive in all cases except once with Lin, where the mean *MPI* slightly increases with careful mapping. The multiprogrammed trace behaviors are all similar, except for Mult1.2, where one bad page placement in one simulation biased the results and cut the direct-mapped Hierarchical reduction in about half (for the 1-megabyte and 4-megabyte caches). Careful mapping doesn't help Tv much because most of its misses occur while it traverses a large data structure with little locality; it cannot eliminate many misses during this traversal because the memory locations tend to be randomly accessed, and the Tv address space is too large for much static placement improvement. (Figure 3 shows how there is little room for static mapping improvements when the active address space size is large relative to the cache size.) The improvements

13. The statistical significance of these and many more results can be checked by consulting Kessler's thesis [KESS91]. In most cases, our results are accurate to within $\pm$5%.

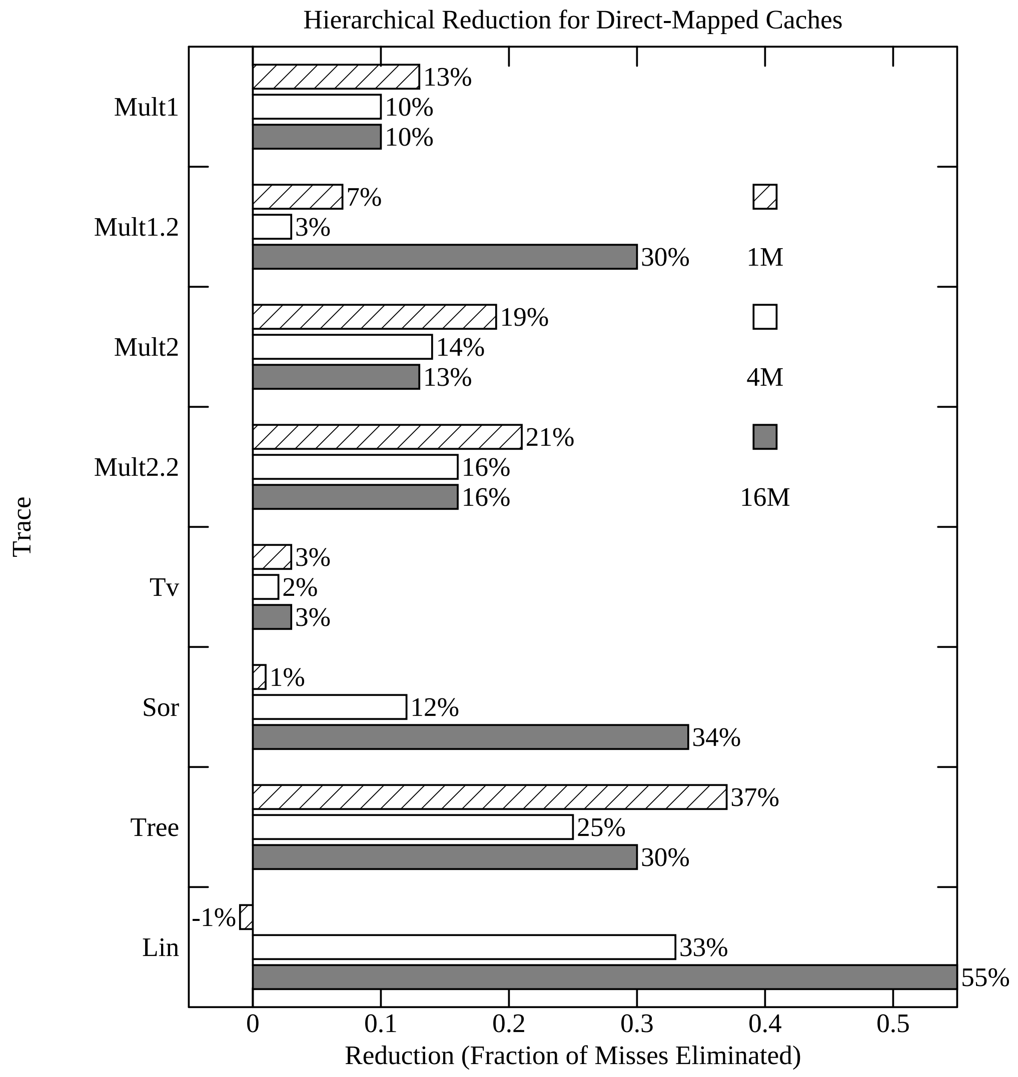

Figure 6. Direct-Mapped Hierarchical Miss Reductions for All Traces.
This figure shows, for different direct-mapped secondary caches and traces, the Hierarchical reduction, which is the fraction of the misses (produced from random mapping) that Hierarchical could eliminate.

of both Sor and Lin relate closely to the benefits of associativity. There is little careful mapping or associativity improvement with the 1-megabyte cache, but both careful mapping and associativity improve performance greatly for the 16-megabyte cache because some arrays fit fully in the cache. Though the results in Figure 6 vary greatly for the different traces, the rest of this study examines Mult2.2 in detail because of its ‘‘average’’ behavior; the Mult2.2 Hierarchical reductions are close to the average of all the traces.

Figure 7 shows the Mult2.2 results in another graphical way to put the careful mapping miss reduction in its proper perspective. By interpolating between the simulation results for different cache sizes and associativities, it estimates the effect of software careful mapping relative to hardware cache design changes. The dashed line in the left graph finds the equivalent associativity change of Hierarchical with a 1-megabyte direct-mapped cache. Careful mapping reduces misses by more than half as much as a random-mapping associativity increase does; it is about equivalent to an associativity increase from direct-mapped to 1.6-way with Mult2.2. For a direct-mapped cache, the dashed line in the right graph compares the careful mapping miss reduction (with a 1-megabyte cache) to the reduction from a randomly-mapped cache size increase. Careful mapping also removes nearly as many misses as a direct-mapped cache size doubling does; it makes a 1-megabyte direct-mapped cache act about like a 1.7-megabyte direct-mapped cache for Mult2.2.

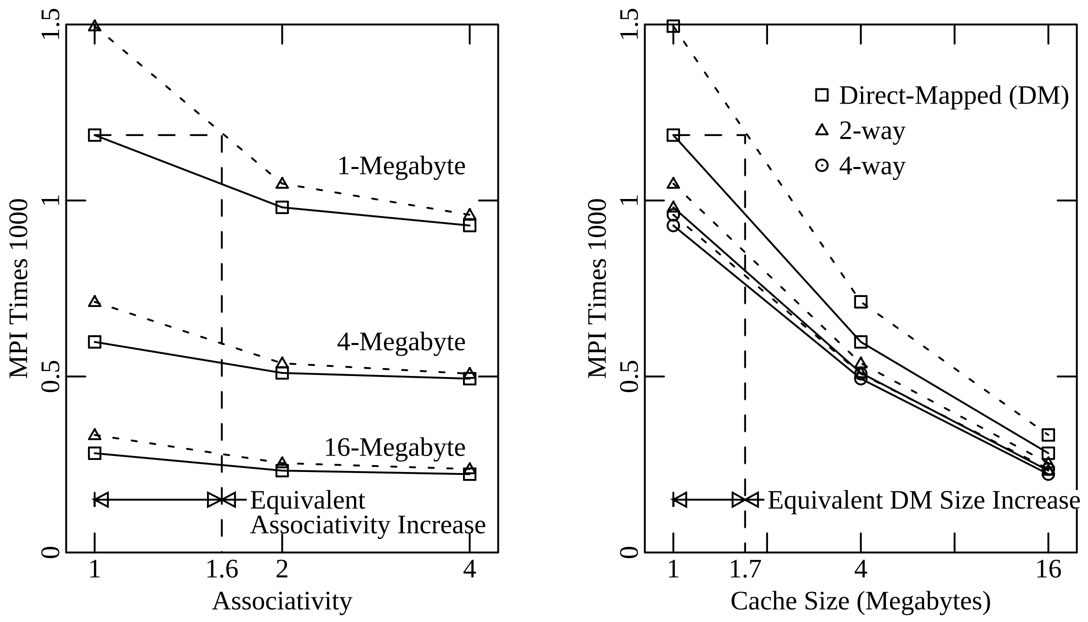

Figure 7. Mult2.2 Careful Mapping Improvement.

This figure shows the (mean) Mult2.2 simulation data. The misses per thousand instructions is plotted versus the cache associativity and size for random (dotted) page mappings and Hierarchical (solid) page mappings. The left graph shows *MPI* versus associativity for the different cache sizes. The right graph shows *MPI* versus cache size for different associativities (square = direct-mapped, triangle = two-way, and circle = four-way)

Most importantly, as we have shown, careful page mapping reduces mean *MPI* (across different page mappings) by avoiding poor page mappings. But a secondarily useful trait of careful mapping is a reduction in the *MPI* variance for different runs of a trace. Variance in cache behavior is undesirable because it increases the execution time variance of a workload. Figure 5 shows that the Mult2.2 mappings produced carefully have less *MPI* variations than the random mappings because all of the solid (Hierarchical) lines are much closer together than the dotted (random) lines. The different careful mapping simulations tend to follow an outline that the workload dictates, but the random mapping simulations show the bursts that are caused by random contention for cache locations. Poor random mappings into the cache, not the workload references themselves, cause many of the random mapping misses. We found that when Hierarchical removes the most misses, it also tends to eliminate the most variance. This isn't surprising since most of the *MPI* reduction comes from eliminating bursts of misses that increase both the mean and variance of the *MPI*. Hierarchical had a lower four-sample *MPI* variance than random mapping for 57 out of the 72 cases we examined. With fourteen out of the twenty-four direct-mapped caches, the random mapping four-sample variance is more than five times larger than Hierarchical. This strongly suggests that Hierarchical will yield more predictable execution times than random mapping.

5.2. Careful Mapping Alternatives

In this section, we compare the Heuristics from Section 3 using the Mult2.2 trace. The size of the available page frame pool has an important affect on the comparison. The implementations that produce the fewest misses with the least available page frames are preferred because they are adaptable to the most situations. For the Hierarchical results given so far in this study, the pool is 4-megabytes. As an experiment to learn the impact of the available pool size, Table 1 compares the 4-megabyte direct-mapped Mult2.2 *MPI* of the different heuristics (four-sample mean) with a 4-megabyte pool of available pages (large) and a 256-kilobyte pool (small)[14]. On the average there was one available page frame in each bin with the large pool, and 0.0625 available page frames in each bin with the 256-kilobyte pool.

Secondary 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%)
Best Bin	0.60 (0%)	0.61 (-8%)
Hierarchical	0.60 (0%)	0.66 (0%)

Table 1. 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. It shows the four-sample mean *MPI* for 4-megabytes of available page frames (Large Pool) and 256-kilobytes of available page frames (Small Pool). In parenthesis, it also gives the relative difference from the Hierarchical results for each pool size. The results shown are for a 4-megabyte direct-mapped cache.

14. Most of these results are accurate to within $\pm$ 0.02 or less.

Equality Page Coloring performs poorly in the Mult2.2 comparison presented in Table 1. Its particularly high mean *MPI* for the small pool comes from the exceptionally high *MPI* of two (out of the four) simulations. This is a symptom of the basic problem with equality page coloring: the placement of commonly used virtual pages from different address spaces in the same cache bins often leads to many unnecessary cache misses. Probably because of this problem, MIPS does not use the strict equality match Page Coloring simulated here. The PID-hashed Page Coloring implementation performs much better for Mult2.2, yet it still performs poorly compared to the other careful mapping schemes. The limited flexibility of the Page Coloring implementation is the cause of this poor performance: if Page Coloring does not find an available page frame in the matching bin, it abandons the search and chooses any arbitrary page frame. With the small pool, an exact match is unlikely (probability 0.0625), so the placement is almost random. The other careful mapping schemes are more adaptable to situations with few available page frames.

The data in Table 1 shows that Best Bin produces superior placements because a smaller pool doesn't significantly affect its performance, while it causes the four simulations with the other heuristics to have more cache misses. With the 4-megabyte pool of available page frames in Table 1, Bin Hopping, Best Bin, and Hierarchical perform nearly equivalently. Each of them finds it easy to reduce cache contention when there is one available page frame in each bin on average. With 0.0625 available page frames per bin, Best Bin produces the fewest misses. This shows that Best Bin is the most flexible across all pool sizes, and that it can improve Mult2.2 cache performance under a wide variety of page-availability conditions.

We only show results for a 4-megabyte direct-mapped cache in Table 1. We also found, when using a single placement to simulate many cache sizes, that Hierarchical eliminated the most misses over the range of sizes. This verifies the size-independence advantage of Hierarchical.

6. Conclusions

When a real-indexed cache is large, the virtual to real page mapping function and the cache mapping function interact to determine cache performance. This gives the page mapping policy the opportunity to improve cache performance by carefully choosing the virtual to real page mapping of an address space. This paper shows the usefulness of careful virtual to real page mapping policies, describes several practical careful mapping implementations, and shows their resulting cache miss reduction using trace-driven simulation. It is worthwhile to expend a little effort to map a page carefully since it can improve memory system performance on all following accesses to the page.

We feel that careful virtual to real page mapping is a useful, low-cost, technique to improve the performance of large, real-indexed CPU caches in many circumstances. We introduced careful mapping algorithms that require no page replacement policy changes: they never force a page replacement when there are page frames in the available page frame pool, and the contents of the available page frame pool are determined solely by the operating system's page replacement algorithm. Our algorithms provide the opportunity for cache performance improvements when the operating system makes more pages available for placement. When main memory is under-utilized, careful mapping algorithms are a cheap way to improve cache performance because it is easy to maintain a large available pool. When main memory is more highly-utilized, only a small pool will be available so careful mapping will be less effective (and less needed because page swapping may dominate performance). An increase in the available page frame pool size gives the careful mapping algorithms more freedom, perhaps with little or no corresponding page fault rate increase since a modest-sized pool provides sufficient flexibility. (Unfortunately, our traces were not long enough to quantify the relationship between page fault rate and pool size.)

This paper shows that static cache bin placement optimizations can produce dynamic cache performance improvements. A simple static analysis shows that random page placement causes many of the pages from an address space to be unnecessarily in conflict, so there is potential for static mapping improvements with careful mapping. This analysis also correctly predicts that conflicts are less of a problem in caches of higher associativity.

This paper describes Page Coloring and introduces several other practical careful page mapping heuristics that improve the cache bin placement of pages based solely on static information. These algorithms equalize cache bin utilization and eliminate much unnecessary cache contention. They range from the simple Page Coloring and Bin Hopping to the more sophisticated Best Bin and Hierarchical. This study quantitatively compares the *MPI* improvements of the heuristics with trace-driven simulations of a multiprogrammed trace (Mult2.2). This comparison shows that Bin Hopping, Best Bin, and Hierarchical perform well. Under this comparison, equality match Page Coloring performed poorly because it maps frequently used areas of the virtual memory space from different address spaces to the same cache bin. A PID-hashed version of Page Coloring that fixes this problem still does not compare favorably to the other careful mapping schemes. We found that Best Bin produces the best page placements with the smallest pool of available page frames; it is more adaptable across the range of possible page-availability conditions. Hierarchical has good computational efficiency and cache size independent mappings; these properties are desirable, particularly for hierarchies of real-indexed caches.

This paper presents simulation results from several workloads to measure the improved *MPI* mean and variance of careful page mapping (Hierarchical). The improvement depends on the workload, but with our traces we found that careful mapping reduced the direct-mapped cache misses per instruction by about 10-20%. Thus, at no hardware cost, software careful page mapping gets about half the improvement of either doubling the cache size or associativity.

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 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 to Harold Stone for his useful comments. The referees also greatly improved the presentation in this paper.

8. Bibliography

[AGHH89] A. AGARWAL, M. HOROWITZ and J. HENNESSY, ''An Analytical Cache Model,'' *ACM Transactions on Computer Systems*, vol. 7, no. 2, May 1989, pp. 184-215.

[BABJ81] O. BABAOGLU and W. JOY, ''Converting a Swap-Based System 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 Symposium on Computer Architecture*, 1990, pp. 270-279.

[BRLF90] B. K. BRAY, W. L. LYNCH and M. J. FLYNN, ''Page Allocation to Reduce Access Time of Physical Caches,'' Technical Report CSL-TR-90-454, Computer Systems Laboratory, Stanford University, Stanford, CA, November 1990.

[DENN68] P. J. DENNING, ''The Working Set Model for Program Behavior,'' *Communications of the ACM*, vol. 11, no. 5, May 1968, pp. 323-333.

[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.

[HILS89] M. D. HILL and A. J. SMITH, ‘‘Evaluating Associativity in CPU Caches,’’ *IEEE Transactions on Computers*, vol. 38, no. 12, December 1989, pp. 1612-1630.

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

[KAPW73] K. R. KAPLAN and R. O. WINDER, ‘‘Cache-Based Computer Systems,’’ *IEEE Computer*, vol. 6, no. 3, March 1973, pp. 30-36.

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

[KESH90] R. E. KESSLER and M. D. HILL, ‘‘Miss Reduction in Large, Real-Indexed Caches,’’ Computer Sciences Technical Report #940, University of Wisconsin-Madison, June 1990.

[KESS91] R. E. KESSLER, ‘‘Analysis of Multi-Megabyte Secondary CPU Cache Memories,’’ Ph.D. Thesis, Computer Sciences Technical Report #1032, University of Wisconsin, Madison, WI, July 1991.

[LARW91] M. S. LAM, E. E. ROTHBERG and M. E. WOLF, ‘‘The Cache Performance and Optimizations of Blocked Algorithms,’’ *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 63-74.

[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.

[NIEL86] M. J. K. NIELSEN, ‘‘Titan System Manual,’’ Research Report 86/1, Western Research Laboratory, Digital Equipment Corporation, Palo Alto, CA, September 1986.

[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.

[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.

[THIS87] D. THIEBAUT and H. S. STONE, ‘‘Footprints in the Cache,’’ *ACM Transactions on Computer Systems*, vol. 5, no. 4, November 1987, pp. 305-329.

[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.