Reducing Cache Misses Using Hardware and Software Page Placement

Timothy Sherwood Brad Calder Joel Emer

Department of Computer Science and Engineering
University of California, San Diego
{sherwood,calder}@cs.ucsd.edu

Alpha Development Group
Compaq Computer Corporation
emer@vssad.hlo.dec.com

Abstract

As the gap between memory and processor speeds continues to widen, cache efficiency is an increasingly important component of processor performance. Compiler techniques have been used to improve instruction and data cache performance for virtually indexed caches by mapping code and data with temporal locality to different cache blocks.

In this paper we examine the performance of compiler and hardware approaches for reordering pages in physically addressed caches to eliminate cache misses. The software approach provides a color mapping at compile-time for code and data pages, which can then be used by the operating system to guide its allocation of physical pages. The hardware approach works by adding a page remap field to the TLB, which is used to allow a page to be remapped to a different color in the physically indexed cache while keeping the same physical page in memory. The results show that software page placement provided a 28% speedup and hardware page placement provided a 21% speedup on average for a superscalar processor. For a 4 processor single-chip multiprocessor, the miss rate was reduced from 8.7% down to 7.2% on average.

1 Introduction

A great deal of effort has been invested in reducing the impact of cache misses on program performance. As with any other latency, cache miss latency can be tolerated using compile-time techniques such as instruction scheduling, or run-time techniques including out-of-order issue, decoupled execution, or non-blocking loads. It is also possible to reduce the latency of cache misses using techniques that include multi-level caches, victim caches, and prefetching.

Many approaches have been examined to eliminate cache misses. Hardware techniques include set-associative caches [23], pseudo-associative caches [1, 5], group-associative caches [27], page coloring [22], predicting which data not to cache [18, 33], and providing conflict miss hardware with operating system support to move pages [2]. Software techniques include program restructuring to improve data [6, 26, 7, 25, 29] or instruction cache performance [12, 15, 16, 24, 28], and compiler-directed page coloring for multiprocessors to eliminate 2nd level cache misses for arrays [3].

When performing placement of instructions and data, the software approaches have shown to eliminate a significant amount of cache misses for virtually indexed caches. For a physically indexed cache, the operating system needs to provide support for page coloring or compiler directed page placement, otherwise it is left to chance which virtual pages will overlap in the physically indexed cache. This can potentially lead to severe conflicts, which could have been otherwise averted with careful placement.

Published in the Proceedings of the International Conference on Supercomputing, June 1999.

In this paper, we examine both software and hardware techniques for performing page placement to eliminate cache misses for a physically indexed 2nd level cache. All of the techniques that we examine leave the virtual address space completely untouched in order to allow prior approaches to perform their best placement, eliminating as many first level virtually indexed cache misses as possible.

In performing this research, we break the 2nd level cache up into N *colors*, where N is equal to the (number of cache sets * block size) / page size. Intuitively a color is a page sized chuck (group of sets) of the cache, where all accesses to a given page will be of the same color. Therefore, two pages map to the same color if they have the same location in the physically indexed cache. We examine automated methods of mapping virtual pages to colors to reduce cache misses. This coloring can then be used by the operating system to allocate pages or by the hardware to re-map physical pages in the 2nd level cache.

Our *software page placement* algorithm performs a coloring of virtual pages using profiles at compile-time. A physical page color is produced for each code and data virtual page used during execution, to reduce 2nd level cache misses. This mapping table is then passed to the operating system when the program starts executing. The operating system uses this mapping as a hint during page allocation in order to place virtual pages into physical pages of a given color as indicated by the compile-time mapping.

Our *hardware page placement* approach for page placement weakly decouples the 2nd level cache from the physical pages. This is done by providing a remapping of where a physical page can be found in the 2nd level cache through a page remapping field stored in the TLB. This field is used as part of the index into the 2nd level cache instead of the physical page number. The triggering of a remapping is done by a small buffer of counters, to keep track of high miss cache sets. When a set accrues enough misses relative to its references, the hot pages using that set are remapped to eliminate cache conflicts. The physical pages do not move in memory, instead, only the location (color) of the page in the 2nd level cache is changed.

The remainder of this paper details the design, implementation, and analysis of page placement. Section 2 motivates the approach by graphically demonstrating the 2nd level cache utilization and the improvement from page placement. Section 3 describes work related to page placement. Section 4 describes the methodology used to gather the results for this paper. Section 5 describes and provides results for the software placement algorithm. Section 6 describes and provides results for the hardware page placement architecture.

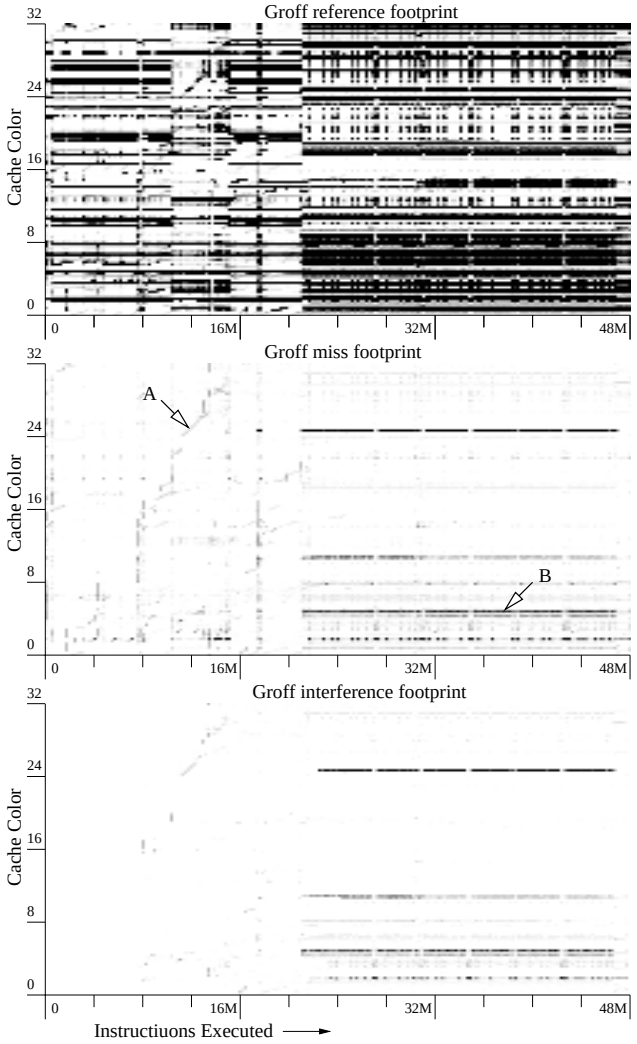

Figure 1: Cache footprints for Groff. The first graph shows the density of L2 cache references during execution, the second graph the density of misses, and the last graph the misses caused between code and data pages. The darker the graph is the greater the number of references/misses to that group of sets during execution. At point A on the miss footprint, striding behavior can be seen. Point B is an example of the striped misses indicative of high conflict sets.

Section 7 provides results for using the hardware placement for a single-chip multiprocessor. Finally, Section 8 summarizes the results and contributions of this work.

2 Motivation

To show why page placement is needed and works we will first examine cache set usage during a program's execution. Figure 1 shows the memory *footprint* for `groff` (a troff text formatter written in C++) for a direct mapped 256K L2 cache with 32-byte lines. Results are shown when breaking the cache up into sets grouping them into 8K pages, which results in 32 colors. A color represents a *group* of sets. The X-axis represents the execution of the program over time in terms of number of instructions executed shown in millions. The Y-axis shows the different colors (sets) in the L2 cache. Four lines are shown for each color, so each line represents 2K of a page. The darker the line is the greater the number of references/misses to that group of sets during execution. Page allocation was performed using Bin Hopping described in section 3.1.

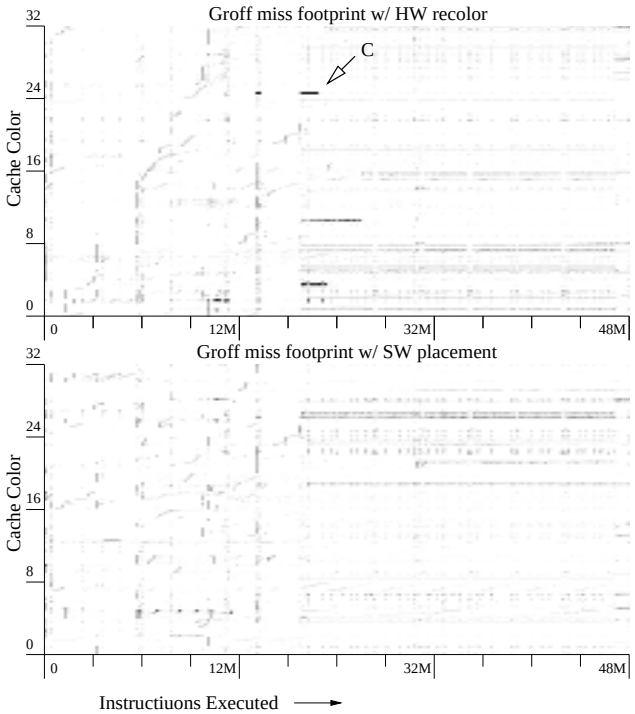

Figure 2: The improved cache footprints for groff using hardware and software page placement. Point C is where a recolor occurs and cache conflicts are removed.

The first result to observe from these graphs is `groff's` overall memory behavior. The first graph in Figure 1 shows the number of references to each set/color over time. The results show that some sets are not used at all, while others are accessed extremely frequently. The latter of these are termed *hot* sets.

In the first part of the execution, `groff` spends some time striding through memory, and then around 16 million instructions the memory usage for `groff` converges to a consistent memory pattern for the rest of the program's execution. We found similar memory behavior for several of the other programs we examined. Adaptive page placement will perform very well for this type of memory behavior.

The second graph in Figure 1 shows the number of misses to each set over time. For `groff` there are only a handful of hot sets with many misses, and these same sets had a high miss rate starting around 16 million instructions through the end of execution. Point `A` in this Figure shows the program striding through memory, whereas point `B` points to a hot set with a high miss rate.

The third graph in Figure 1 shows only those misses that are caused between the interference of instruction fetch misses and data misses. As can be seen, many of the hot miss sets are caused because of the interference between code and data pages.

Figure 2 shows the cache usage over time for `groff` when using page placement described in Sections 5 and 6. The first graph shows the misses when using hardware placement, and the second graph when using software guided page placement. The hardware placement graph shows four distinct places where the page placement was performed. One place can be seen at Point `C`, where after placement color 24 no longer has a high miss rate. Comparing this graph to the miss graph in Figure 1, shows that the number of high conflict sets (dark lines) has decreased significantly and that the usage of the cache is spread out more evenly over all the cache sets. Similar results are seen for the software page placement results.

3 Related Work

There has been much research in the area of code, data, and page placement to improve memory hierarchy performance. In this section, we concentrate on prior work related to program placement to reduce cache misses for physically indexed caches.

3.1 Operating System Page Allocation

The research by Kessler and Hill [21] represents an extensive examination of different operating system page placement algorithms and their performance. They examined several mapping algorithms, and found Page Coloring and Bin Hopping to provide good performance.

Page Coloring maps consecutive virtual pages to consecutive colors, and is used by Windows NT. Each virtual page modulo (the number of cache sets * the block size) maps to the same color in a physically indexed cache. Physical page allocation is performed by the OS to maintain this mapping. This allows the compiler to potentially use code and data placement techniques to help eliminate cache misses by mapping data with temporal locality that should not be put on the same page to different virtual pages with different colors.

Bin Hopping allocates pages to sequential colors in the order that page faults occur. This allows pages that are touched and cause page faults close in time to map to different locations (colors) in the cache. The Bin Hopping algorithm is used by existing operating systems such as Digital Unix (OSF) for allocating pages. We use Bin Hopping as our default page allocation algorithm for our baseline configuration.

3.2 Software Guided Page Placement

Custom operating systems, such as Exokernal [10] and V++ [14] have been designed that allow applications to provide their own page replacement and page mapping policies.

Bugnion et. al. [3] recently examined using compiler directed page coloring for arrays on multiprocessors. Their approach at run-time generates a preferred coloring for data pages containing arrays. The coloring is generated using compiler generated analysis for the access patterns and the array sizes provided at run-time. This coloring is then used as a hint to the operating system when it performs its coloring. Our software guided placement extends their research by applying the compiler directed approach to all data and code pages instead of just to arrays by using profiles.

3.3 Hardware Support for Page Placement

The Impulse project provides a compiler controlled memory controller [8]. The Impulse address space can be remapped by providing a new strided address calculation or an indirection vector for an array. All remapped data accesses go through this address translation to compute the real location of the data. This can be very efficient for optimizing sparse matrices, arrays with stride access, and placing arrays in a given location in the L2 cache. Yamada et. al. [35] proposed a similar approach for compiler controlled memory layout, but for the first level cache. Our approach is simpler, and requires less hardware. We only concentrate on providing page remapping. One benefit is that our approach does not require the extra stage of translation needed in the above work to find the remapped data. Instead, our approach uses an extra field stored in each TLB entry, which is used in the normal TLB translation to determine where to locate the page in the physically indexed L2 cache.

Bershad et al. [2] examined using a small Cache Miss Lookaside (CML) buffer that detects conflicts by recording the number of cache misses to frequently referenced pages. Romer et. al. [30] extended this work using counters and entries in the TLB to find conflicting pages instead of the CML. The recoloring used in both of these papers traps to the OS, which does a full copy of a page to the new physical location to perform the recoloring. Our hardware page placement approach is very similar to the CML approach, except we do not move pages around in the physical address space. Instead, we change the mapping (color) of the page in the physical indexed L2 cache in order to move the page as described in section 6.

In the next section we will describe our experimental methodology followed by our software and hardware page placement strategies.

4 Methodology

To perform our evaluation, we collected information for 6 of the SPEC95 benchmarks, and two C++ programs `groff` (a troff text formatter) and `deltablue (db++)` (a constraint solution system). The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C, C++, and FORTRAN compilers. We compiled the programs under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). We used ATOM [31] to instrument the programs when gathering the page reference profiles for software page placement.

The simulators used in this study are derived from the SimpleScalar/Alpha 2.1 and 3.0 tool set [4], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions, performing a detailed timing simulation of an aggressive dynamically scheduled microprocessor with two levels of instruction and data cache memory. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. The baseline microarchitecture model is detailed in Table 1.

Our baseline simulation configuration models a future generation microarchitecture. We've selected the parameters to capture two underlying trends in microarchitecture design. First, the model has an aggressive fetch stage, employing a variant of the collapsing buffer[9]. The fetch unit can deliver two basic blocks from the I-cache per fetch cycle up to 8 instructions. Second, we've given the processor a large window of execution, by modeling large reorder buffers and load/store queues.

We modified SimpleScalar to model a complete memory hierarchy, with page allocation and bus contention. We modeled an on-chip/near-chip L2 cache with a 6 cycle latency to access a direct mapped L2 cache and a 7 cycle latency to access a 2-way associative L2 cache. An L2 miss has a 180 cycle latency for retrieving data from main memory.

Table 2 shows the two inputs used in gathering results for each program. The first input *train* was used to gather the page reference profiles for software page placement. The second input *test* was used to gather all the simulation results. Each program was simulated for 100 million committed instructions plus the number of instructions (in millions) shown in the Fast-Fwd column. The detailed IPC and cache miss results were only gathered for the 100 million instructions after fast forwarding over initial startup code. The Base IPC column shows the IPC when using a direct mapped 256K L2 cache with 32-byte lines, and this is used as our baseline configuration. The next column shows the number of unique static 8K pages used during our simulation of each program. The *Code* column shows the percent of these static virtual pages that were code pages. The next two columns show the percent of misses to the L1 instruction and data cache. The last column shows the number of references to the L2 cache in millions.

Fetch Interface	delivers two basic blocks per cycle, but no more than 8 instructions total
Instruction Cache	32k 2-way set-associative, 32 byte blocks, 6 cycle miss latency
Branch Predictor	hybrid - 8-bit gshare w/ 16k 2-bit predictors + a 16k bimodal predictor 8 cycle mis-prediction penalty (minimum)
Out-of-Order Issue Mechanism	out-of-order issue of up to 16 operations per cycle, 512 entry re-order buffer, 256 entry load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	16-integer ALU, 8-load/store units, 4-FP adders, 1-integer MULT/DIV, 1-FP MULT/DIV
Functional Unit Latency (total/issue)	integer ALU-1/1, load/store-2/1, integer MULT-3/1, integer DIV-12/12, FP adder-2/1 FP MULT-4/1, FP DIV-12/12
Instruction Cache Data Cache Virtual Memory	32k direct mapped, write-back, write-allocate, 32 byte blocks, 6 cycle miss latency 32k 2-way set-associative, write-back, write-allocate, 32 byte blocks, 6 cycle miss latency four-ported, non-blocking interface, supporting one outstanding miss per physical register 8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table 1: Baseline Simulation Model.

Program	Train	Test	Fast Fwd	Base IPC	# Pages	%Code	32K-I	32K-D	L2 Refs
groff	paper-me	someman	25M	1.49	74	64.9%	2.1%	<0.1%	0.8 M
delta-blue	train	ref	9M	2.34	458	2.1%	0.6%	8.9%	2.1 M
compress	short	ref	0M	3.21	97	4.0%	<0.1%	5.2%	0.9 M
go	2stone9	5stone21	5M	2.53	85	70.2%	0.5%	0.2%	0.9 M
gcc	1recog	1cp-decl	75M	1.24	515	32.5%	1.1%	1.0%	2.2 M
m88ksim	train	ref	325M	3.20	1332	27.9%	0.9%	0.1%	1.2 M
vortex	train	ref	12M	1.58	913	26.6%	2.0%	1.3%	2.7 M
tomcatv	train	ref	425M	1.69	71	37.9%	0.9%	0.1%	1.3 M

Table 2: General Program Statistics.

For the superscalar results we modeled a 256K L2 cache because the memory footprints of our benchmarks are smaller than heavily used applications such as MS Office and commercial databases. This can be seen in Table 2 where the number of pages used by 5 of our programs fits entirely within 1 Meg of memory, and the remaining 4 programs use from 4 Meg to 10 Meg of memory.

5 Software Page Placement

In order to allow software page placement, the compiler must have some control over the virtual to physical mapping. In addition, it has to have some cooperation from the operating system, which is ultimately in control of the mapping. Previous work described an implementation that allows a program at run-time to communicate hints directly to the OS for which colors (in the 2nd level cache) virtual pages containing arrays should be mapped to [3]. We assume a similar type of support for our software page placement, except the color mapping is generated at compile-time using profiles for all virtual code and data pages. Remember that the number of colors in the L2 cache is N, where N is the (number of sets in the L2 cache * block size) / page size. Therefore, the physical pages are broken up into N colors.

The first step of our software page placement algorithm is to use profiling to create a *Temporal Relationship Graph* (TRG) describing which pages are referenced near each other in time. Pages with temporal relationships should avoid being placed in the same color in order to eliminate cache misses. After the TRG profile is created, all the virtual pages that were executed are assigned a color using a greedy algorithm. After the coloring, a static array is created that contains a mapping for all the popular virtual page numbers and their corresponding colors. Virtual page numbers that were not referenced enough use the default coloring of the operating system. When the program starts executing, it passes this color mapping to the OS. The OS then keeps this mapping and uses it as a hint for coloring when a page fault occurs.

The TRG and coloring algorithm used in this paper are derived from our prior research into code [12] and data [6] placement to eliminate first level cache misses. In the rest of this section we describe these steps in more detail.

5.1 Building a Page Temporal Relationship Graph

Profiling is used to keep track of possible page conflicts by building up a temporal relation graph during the execution of each of the programs. A *relationship* bit matrix of size $P \times P$, where P is the number of pages, is used to find relationships between pages. A second matrix, the *conflict* matrix, also of size $P \times P$, is used to keep track of the number of relationships between pages.

The relationship matrix is used to find a temporal relationship between two page references A and B of the form $A \rightarrow B \rightarrow A$, where $\rightarrow$ denotes sequential ordering of references to the pages A and B. A reference to a page is any access to a code (instruction fetch) or data (load or store) page. When a reference to a page B occurs between two references of another page A, then the second reference to A can result in a cache miss if B and A are mapped to the same color. We keep track of relationships at the level of 1/4 the size of an 8K page rather than at the page level or at the cache block. We examined keeping track of the relationships at smaller granularities, and it provided no significant benefits for the programs examined.

On each reference to a page A, all of the bits in the relationship matrix on *column* A are set to 1, except (A,A). Next, all of the bits in *row* A are checked for a conflict in the relationship matrix. For each column X that has a bit set for row A in the relationship matrix, a counter is incremented at (A,X) in the conflict matrix to represent an interference edge between A and X. The only exception to this is the matrix element (A,A) which is ignored because it represents self interferences which are not particularly useful. The last step is

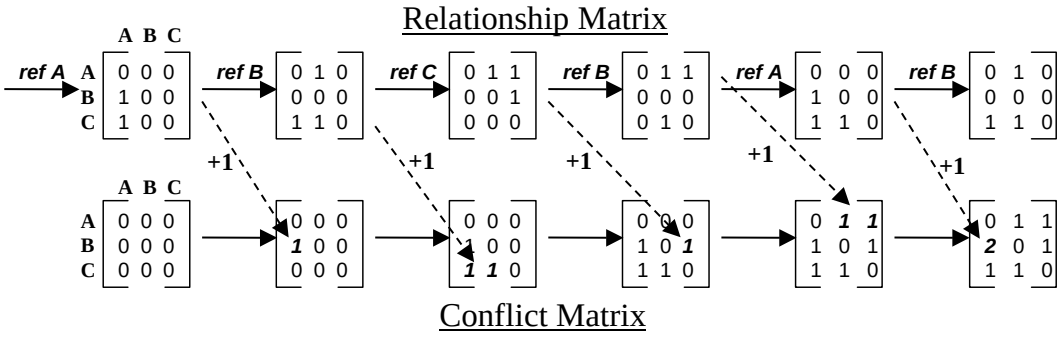

Figure 3: An example matrix for the reference stream: A,B,C,B,A,B. The top matrix row shows the relationship matrices, and the bottom row shows the values in the conflict matrix. The matrix to the right of each reference is the matrix after performing the updates from the corresponding reference.

to clear all the bits in row A in the relationship matrix.

Figure 3 shows an example of the profiling algorithm for reference stream $A \rightarrow B \rightarrow C \rightarrow B \rightarrow A \rightarrow B$, where A, B, and C are pages. The top row of arrays are the relationship matrices, and the bottom row are the corresponding conflict matrices for each page reference. After the first reference to A the bits in the column for A are set and the conflict matrix is initially all zeros. When referencing B, the row B is checked for set bits. A set bit is found at (B,A) in the relationship matrix. Therefore, the conflict matrix element (B,A) is incremented by 1. Next, all the bits in the column for B are set (except (B,B)) in the relationship matrix. Finally, the row B has its bits cleared in the relationship matrix. The second pair of matrices in Figure 3, show the resulting matrices after processing the first reference to B.

The final conflict matrix, once complete, contains for each pair of pages a cost metric denoting how many times they have interleaved their references, potentially causing cache conflicts. Intuitively pages with many interleaving references should not be placed in the same color because the interleaving references might kick each other out of cache frequently yielding high miss rates. The TRG is then used by the placement algorithm to provide a low conflict coloring.

To decrease the profiler's memory usage, when storing the relationship and conflict matrix, only the most frequently referenced pages are tracked when building the TRG. Therefore, we first profile the program to find the pages that account for up to 99% of all the references. Relationships are only kept track of for these pages when building the TRG. This significantly reduced the amount of memory and computation time needed to generate the TRG.

5.2 Producing a Mapping Using Page Coloring

The final conflict matrix is the temporal relationship graph, and it is used to build a coloring of pages to eliminate 2nd level cache conflicts. The matrix is viewed as an undirected graph with pages as vertices and temporal relations as edges. To convert the conflict matrix to a TRG graph, each edge created between two pages X and Y is assigned an edge weight equal to the sum of the two entries (X,Y) and (Y,X) in the conflict matrix.

To create a page placement we must assign each node one of the possible page colors. The total cost of assigning a page a given color is determined by the sum of the relationship edge weights in the TRG for pages already mapped to that color. To minimize total cache conflicts we need to find a coloring with the smallest possible cost. Since the problem is a superset of the k-coloring problem, which has been shown to be NP-complete, a heuristic is used.

The algorithm processes the TRG edges between pages from the largest relationship edge to the least. When processing an edge between 2 pages, either both of the pages will have already been colored, both of the pages will be uncolored, or just one of the pages will be uncolored. Each uncolored page for the edge calculates the cost of assigning each of the colors to that page. The cost of placing a page in a given color is calculated as the total cost of the TRG edges between the uncolored page and all the pages already assigned to that color. The smallest costing color is then assigned to the uncolored page, marking the page as colored. This step is repeated until all the TRG edges have been processed. After a page is assigned its first color, that coloring is final, so a page is not allowed to change colors.

No color is given to the pages that were not profiled, and the infrequently referenced pages. Instead, the default page allocation algorithm will be used for these pages by the operating system. In this research we used Bin Hopping as described in section 3 as the default operating system page allocation algorithm.

5.3 Software Results

We now show SimpleScalar results for the architecture described in section 4 with a 256K L2 cache. We modeled a 256K L2 cache, since most of the programs used less than 1 Meg of total memory during simulation. All the results in the section are shown for simulating the *test* input, and using the *train* input to generate the software page placement as shown in Table 2. This provides realistic results in the sense that different inputs were used to profile the program and a gather the simulation results.

Figure 4 shows the L2 cache miss rates for different associativities with and without software page placement. The results show that there is no improvement in the L2 miss rate for `deltablue`, which allocates and deallocates thousands of objects, and `compress` which has a lot of L2 capacity misses. Other programs like `m88ksim` and `tomcatv` show a significant reduction in miss rate when using software placement. The remaining programs have a lower miss rate from software page placement even when using a 2-way associative L2 cache.

Figure 5 shows the percent IPC speedup over the direct mapped L2 for different L2 associativities with and without software page placement. The results shows that the speedup seen by the direct mapped cache with software page placement is *higher* than the 2-way associative cache for `groff`, `go`, and `tomcatv`, even though the 2-way associative cache has a slightly lower miss rate as seen in Figure 4. The main reason for this is that we simulated a 6 cycle access time for the direct mapped L2 and a 7 cycle access time for the 2-way associative L2 cache to account for the additional delay in performing the block selection.

It is interesting to note that the speedups for `groff` and `vortex`

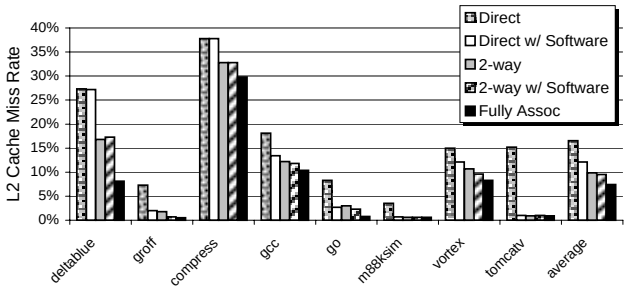

Figure 4: 256K L2 Cache Miss Rate. Results are shown for direct mapped, direct mapped with software page placement, 2-way associative, 2-way associative with software placement, and fully associative.

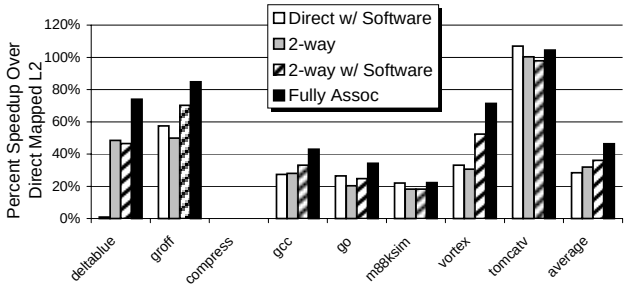

Figure 5: Percent IPC speedup over direct mapped 256K L2 cache. Results are shown for direct mapped with software page placement, 2-way associative, 2-way associative with software placement, and fully associative 256K L2 cache.

are increased by 20% when using software page placement with a 2-way associative cache, when compared to the default 2-way associative cache results. One reason for this is that both `groff` and `vortex` suffer from a large number of L2 misses to code pages. The resulting software page placement reduced the percent of code misses more than the percent of data misses. Data misses can be much more tolerated on an out-of-order processor than code misses. This implies that better results may be possible by giving priority to placing code pages over data pages in the 2nd level cache, which we are examining as part of future research.

6 Hardware Page Placement

The prior section showed that software profiled-guided page placement provides improved cache performance with operating system support. In this section we examine a hardware technique for allowing a processor to make informed decisions as to where pages should be placed in the 2nd level cache and aid in their placement. Our page placement and remapping hardware mechanisms were designed to take up as little chip area as possible and to be easily implementable in hardware, while still providing useful conflict information and resolution. The rest of this section contains a description of each hardware mechanism and how it is used to perform hardware page remapping and placement.

6.1 Hardware Page Remapping

In order to be capable of significantly improving performance by remapping pages at run-time, the cost associated with remapping a page must be sufficiently small. In particular, performing an actual move of data in main memory is clearly not cost effective. The cost

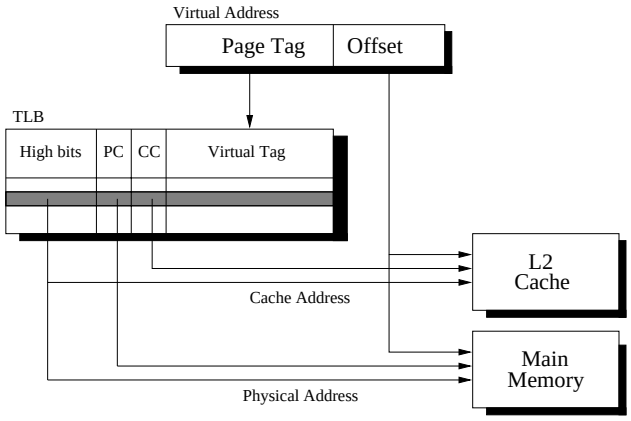

Figure 6: Page remapping hardware. The PC is the Physical Color, where as CC denotes Cache Color. The Cache color bits, which are used instead of the Physical Color bits when accessing the L2 cache, are the only new bits needed in the TLB.

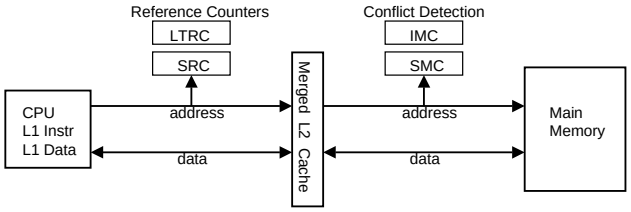

Figure 7: A block model of different pieces of hardware used for dynamic detection of conflicts. The figure depicts the Sampling and Long Term Reference Counters (SRC and LTRC), the Sampling and Interrupt Miss Counters (SMC and IMC).

of going all the way out to DRAM and back for each cache line in a page would limit the applicability of page recoloring to only those interferences that persist through the execution of a large program. To avoid this we propose a modified TLB for fast page recoloring.

A traditional TLB contains fields for the virtual address tag and the physical address. The lower bits of the physical address provide the mapping into the L2 cache. We propose keeping a new field, known as the *cache color*, which can provide an alternate mapping into the L2 cache. When main memory is being accessed, then the original physical address is used. However, when the L2 cache is being accessed, the cache color bits are used rather than the low order bits of the physical address. This means that to perform a recolor the only steps that must be done are changing the cache color bits along with flushing that page from the L2 cache. Figure 6, shows how the addresses are created to index into the L2 cache and main memory.

The three functions necessary for dynamic cache conflict removal are conflict detection, conflicting page recognition, and cache usage. Figure 7 shows the hardware page placement architecture. The conflict detection mechanism informs the processor that there is a cache color/set that is experiencing a lot of misses. The processor then queries the TLB to find the active pages that are mapped to that color. The cache usage is kept track of by reference counters for each color/set and is involved in helping find a new color for the troublesome pages. Table 3 shows the parameters used for the different parts of the architecture, which will be described later.

Number of Colors	32 colors (256K L2 / 8K page)
Number of Counters	128 counters (4 per color)
Sampling miss counter thresholds	128 upper, 64 lower
Interrupt miss counter trigger point	8 (when the counter reaches 8, the processor is informed)
Sampling reference counter thresholds	256 upper, 256 lower
Long term reference saturation point	8 (the reference count cannot be incremented above this value)

Table 3: Hardware Parameters.

6.2 Conflict Detection

The conflict detection mechanism looks for hot sets as discussed earlier in section 2. As described earlier, the cache sets are grouped together into colors based on the size of the page. Conflict counters are implemented in a buffer with an entry for each group of sets (color). For the results in this paper, we provide 4 conflict buffer entries per color. Therefore, each group of 2K cache sets has its own counters.

Each conflict buffer entry has two counters. The first counter is called the *Sampling Miss Counter* (SMC). It keeps track of page misses during each sampling time period, and is cleared every quarter million of executed instructions. The second counter is called the *Interrupt Miss Counter* (IMC), and it accumulates the number of times the SMC is above the sample threshold at the end of the sampling time period.

At the end of the quarter million instruction sampling period, all of the SMC counters are checked sequentially against an increment and decrement threshold. If the SMC counter is above the increment threshold then the corresponding IMC counter is incremented. Conversely if the SMC counter is below the decrement threshold then its IMC is decremented. At the end of the update, all the SMC counters are reset to zero and the process is started again.

A *conflict interrupt* is raised when an IMC counter reaches a conflict threshold value of 8. Misses in hot sets will cause the SMC to be constantly above it's threshold, which will in turn cause the IMC counter to be incremented until it reaches the threshold.

Once a problem set has been detected a conflict interrupt is raised. At this point the OS only knows that there is an L2 cache conflict problem, and it knows the group of sets in the cache causing the problem (i.e., the color of the conflict). To translate this information into a list of actual pages to be recolored, we examine the TLB to find the active pages that are mapped to that color/set.

6.3 Keeping Track of Page References

To decide where to place a page to be remapped we also keep track of two reference counters for each group of 2K sets in the L2 cache. These reference counters are used to find the cold sets to guide which color to remap a hot page too.

The functionality of the two reference counters is similar to the two miss counters described above. The first counter, *Sampling Reference Counter* (SRC), is a saturating counter that counts the number of references during the quarter of a million instruction time interval. After each sampling period, if the SRC is above the reference threshold then the second counter, *Long Term Reference Counter* (LTRC) is incremented. If SRC is below the reference threshold, then LTRC is decremented. When recoloring a page, the LTRC for all the colors are examined to pick a cold color to place the page. After each quarter of a million sampling interval, all the SRC counters are cleared.

Table 3 shows the parameters used for the conflict and reference counters used to gather the results in this paper for a 256K L2 cache with 32-byte lines.

6.4 Page Relocation Algorithm

When page recoloring is triggered by the interrupt miss counter, a conflict interrupt traps to the operating system causing the page relocation algorithm to be invoked. The first step is to find the actual pages that are causing the conflicts. This is accomplished by walking the TLB looking for the pages that are mapped to the high conflict colors.

Only the most recently accessed page in the TLB that maps to the conflicting color is chosen for relocation. If other pages are still causing conflicts, they will be recolored the next time a conflict interrupt occurs. To relocate a page, the algorithm uses the 4 IMC counters of the page's current color to determine the parts of the page that are having the most conflicts. The cost of placing a page in a given color is calculated by taking each of the 4 IMC counters, multiplying them by their corresponding LTRC counters for that color, and then adding together these four calculations. The color with the smallest cost is then chosen for the page.

To remap a page to its new color, we invalidate *all* the cache sets for the old color, moving all of the dirty blocks to the write back buffer. This guarantees that none of the cache blocks for the page we are moving are left behind in the old color. The cache color field in the TLB entry for the page is then updated with the new color. This provides an efficient implementation for the remapping.

The last thing that needs to be done before normal activities can be resumed is to update the hardware counters. To make sure that that a page is given a fair chance at it's new color, the corresponding interrupt miss counters are zeroed for *only* the old color and the new color. In addition, the LTRC reference count for the new color is set to the maximum threshold value of 8. This prevents this new color from being chosen as a cold color if another conflict interrupt is raised in the near future.

When there is no good place to recolor a page, then the conflict detection interrupt is simply ignored. This occurs when there are a lot of capacity misses. This situation is detected when the LTRC counters are saturated for all the colors with a value of 6 or higher in our implementation.

6.5 Implementation Issues

We assume shared pages are not candidates for recoloring. When an application communicates to the OS that a page is to be shared, the page is marked as shared in the page table and TLB, and we do not relocate that page.

When an entry is removed from the TLB, its coloring and the location of its data could be lost. This is taken care of by modifying the page table to contain the cache color field when an entry is evicted from the TLB. Therefore, when a page is brought back into the TLB, it is given its old color back. This has the drawback of increasing the size of each page table entry by 5 bits for a 32 color L2 cache.

The cost of performing a coloring includes the cost of the interrupt and the cycles it takes to recolor each page. We model this in our simulations assuming it takes 200 cycles for each interrupt, and 500 cycles to find a color for each page being recolored. These numbers are similar to those used in prior studies [2, 17, 30], but adjusted for an out-of-order processor.

6.6 Results

We now show SimpleScalar results for the architecture described in Section 4 with a 256K L2 cache. We modeled a 256K L2 cache, since most of the programs used less than 1 Meg of total memory during simulation. Figure 8 shows the L2 cache miss rates for different associativities with and without hardware page placement. There is no improvement in the L2 miss rate for `m88ksim` and `compress`. The interrupt conflict counter threshold for these two

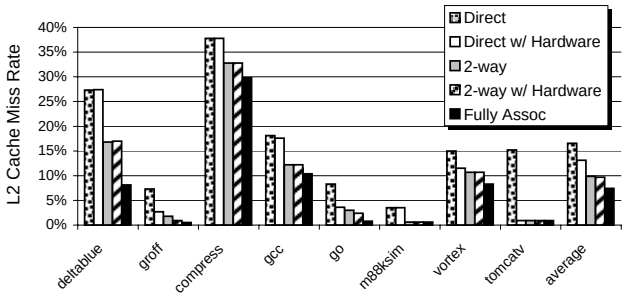

Figure 8: 256K L2 Cache Miss Rate. Results are shown for direct mapped, direct mapped with hardware page placement, 2-way associative, 2-way associative with hardware placement, and fully associative.

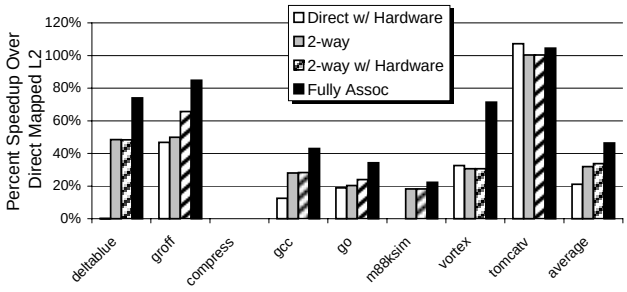

Figure 9: Percent IPC speedup over direct mapped 256K L2 cache. Results are shown for direct mapped with hardware page placement, 2-way associative, 2-way associative with hardware placement, and fully associative.

programs was never achieved, which resulted in 0 conflict interrupts (recolorings), as seen in Table 4. The miss rate for `gcc` did not see the improvement as seen from software placement, because the cache performance was degraded by performing too many recolorings. The rest of the programs showed similar reductions in the miss rate to the software page placement.

Figure 9 shows the percent IPC speedup over the direct mapped L2 for different L2 associativities with and without hardware page placement. Speedups of 0% to 104% are seen for the hardware placement, but they are lower than those seen with software placement. There are three reasons for this. First, the hardware page placement architecture takes time to train to find a page that needs to be recolored. Second, moving a page flushes all the cache sets for the old color causing more cache misses to refill these cache lines. Lastly, since we perform the recoloring in the OS each coloring has to pay the cost of an interrupt and the cycles needed to recolor each page. The benefit of the hardware placement approach over our software placement is that it can adapt to different inputs, and the programs don't need to be profiled to generate the placement.

Table 4 shows cache miss statistics and coloring statistics for the hardware placement for a 256K direct mapped L2 cache. The first column shows the percent of L2 cache misses when both instructions and data are stored in the L2 cache. The second column shows the percent of L2 cache misses if we assume all the references to the data pages are not stored in the L2 cache. The third column shows the percent of L2 cache misses if we assume that no code pages are stored in the L2. In calculating these miss rates the misses are all divided by the total number of potential data and code L2 references. These results show the interference between the code and data pages in the L2 cache. For example, if the accesses to code and data pages for `gcc` would never conflict in the L2 cache, then the miss rate would be 14.7% (11.1% + 3.6%). Instead the miss rate is 18.2% because of the code pages conflicting with the data pages. The remaining two columns in Table 4 shows the number of conflict interrupts raised (number of recolorings), and the number of pages recolored during each interrupt on average for each program.

Program	I and D	Instr Miss	Data Miss	# inter	remaps/inter
delta-blue	27.3%	0.0%	26.9%	28	2.3
groff	7.3%	5.2%	1.1%	5	1
compress	37.8%	0.0%	37.7%	0	0
go	8.3%	6.7%	0.4%	9	1
gcc	18.2%	11.1%	3.6%	127	1.1
m88ksim	3.5%	2.9%	0.6%	0	0
vortex	15.0%	5.4%	7.9%	15	1
tomcatv	15.2%	14.3%	0.9%	733	1.2
Average	16.6%	6.0%	9.8%	115	1.3

Table 4: Hardware Recoloring Statistics.

7 Page Placement for Single-Chip Multiprocessor

In the previous two sections we showed that software and hardware page placement can improve the performance of a wide-issue superscalar processor. Software placement was shown to provide slightly better performance than hardware placement, because of the extra overhead of doing the hardware placement and time needed to find the hot sets, which need to be recolored. Software placement was shown to perform well for eliminating cache conflicts for pages within a given application, but for some future generation processors it may be just as important to eliminate 2nd level cache conflicts between different applications. Hardware placement can potentially perform much better than software placement for these future generation processors, by eliminating conflicts between different applications.

Two interesting choices for future high-end processor designs include Simultaneous Multithreading (SMT) [32] and Single-Chip Multiprocessor (CMP) [13]. Both of these designs have the ability to run several processes concurrently on-chip sharing parts of the memory hierarchy at a fine (instruction) level of parallelism. All the processes running concurrently on an SMT processor share the first level caches and share a combined L2 cache. In the CMP processor, each on-chip processor has their own L1 caches and TLB, but they all share a common L2 cache.

Our hardware approach for eliminating L2 conflicts has the ability to eliminate 2nd level cache conflicts on an SMT and CMP processor by remapping pages eliminating conflicts between different processes executing concurrently.

In this section we examine the miss rate performance for a 4 processor CMP. We used ATOM [31] to simulate a CMP memory hierarchy, where each processor has its own L1 data and instruction cache and TLB, using the same sizes and latencies described in section 4. For the 2nd level cache we simulated a 1 Meg and 2 Meg direct mapped cache. A CMP machine can be used to run parallel programs or several different processes at once, which is more like a server workload and what we modeled. To gather our results we interleaved the execution of 4 different programs and their accesses to the memory hierarchy.

Figure 10 shows the miss rates for the 4 processor CMP for two groups of programs for a 1 Meg and 2 Meg direct mapped L2 cache. The first group of programs contains `groff`, `gcc`, `m88ksim` and `tomcatv` which have a lot of L2 misses due to instructions. The other group `compress`, `deltablue`, `go`, and `vortex` have a high number of data misses. Results are shown for using Page Col-

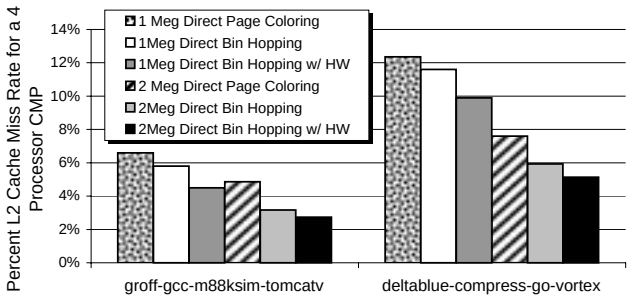

Figure 10: CMP L2 Cache Miss Rate for a 1 Meg and 2 Meg 2nd level direct mapped cache with and without hardware page placement.

oring and Bin Hopping as described in section 3. On a CMP, we modeled a form of global Bin Hopping allocation which allocates pages as faults occur on any of the 4 processors. Global Bin Hopping will most likely be more beneficial than Page Coloring on a CMP for non-parallel workloads, since Page Coloring is mainly used to eliminate misses within a single application. In comparison, global Bin Hopping can potentially help eliminate conflicts between processes as well as within a process. In Figure 10, the 1 Meg results show that the miss rate is reduced from 5.8% down to 4.5% when using hardware placement for the first set of programs, and from 11.6% down to 9.9% for the second set of programs.

8 Summary

In this paper we examined the performance of software and hardware page placement for a superscalar processor, and hardware page placement for a single-chip multiprocessor.

The superscalar results showed that software placement provides a 28% speedup on average for a direct mapped L2 cache, and increased the speedup from 31% to 36% for a 2-way associative cache. The hardware results provided a 21% speedup over a direct mapped L2 cache, and an absolute increase in speedup of 2% when using a 2-way associative cache on average. The best performance was seen for `groff`, where an additional 20% speedup was seen when using either software or hardware page placement for a 2-way associative cache.

For the Single-Chip Multiprocessor, we showed that global bin hopping across all processes for a 1 Meg L2 cache had a miss rate of 8.7%. This performed better than the average miss rate of 9.5% for Page Coloring. When using hardware page placement the miss rate was reduced down to 7.2% on average.

We showed that page placement is important especially for eliminating conflicts between code and data pages. Effective page placement will be of increasing importance as memory latencies grow and future processor designs like CMP and SMT put increased stress on the shared 2nd and 3rd level caches.

8.1 Page Placement vs Increasing Associativity

There are several situations when page placement may be advantageous over simply increasing cache associativity. The most notable are speed, use of commodity parts for off chip caches, and power.

One reason for using page placement over high associative caches is speed. As cache associativities increase, both their cycle and access times increase [34]. Therefore, decreasing the miss rate using page placement without adding additional associativity could provide overall better performance.

Off chip L2 and L3 caches will most likely be direct mapped. One reason for this, is the use of commodity SRAMs for off chip caches. In order to make commodity SRAMs act as associative caches the associativity class (the way) needs to be determined before it can proceed with the cache access. This serialization of tag lookup with cache access can increase the total access time by a significant amount, which makes a direct mapped design attractive for off chip caches. Applying hardware and software page placement for these physically indexed off chip caches can be very beneficial.

Another compelling reason to use a low associative cache is for reduced power consumption. Power consumption increases between 25% to 50% with an increase in associativity [11, 20]. This is of serious concern in many low power designs, such as the StrongARM, because up to 50% of the total processor power is dissipated in the cache. To reduce power usage, the cache associativity needs to be kept low. A conflicting design goal is to also keep the miss rate low in order to save power, since a cache miss will drive another level of memory, consuming more power. Page placement can be used to help reduce the number of misses for these low associative cache designs.

8.2 Future Directions

For future work we plan on examining the performance effects of combining code [12, 15] and data placement [6, 26] techniques for virtually indexed first level caches in combination with the page placement techniques presented in this paper. In addition, we plan to compare the performance of page placement to other hardware techniques (e.g., victim caches [19]) used to reduce the miss rate for low associative caches.

One of the results we found in this study was the importance of placing code pages over data pages. Latencies due to data misses are much easier to mask in an out-of-order processor than code misses. One way to improve our page placement algorithm is to give priority to placing code pages. This would further reduce the number of misses for code pages at the sacrifice of allowing more data misses. Another potential optimization is to enhance the replacement policy used for unified caches to give priority to cache blocks containing code over cache blocks containing data. Another possible optimization is to provide separate victim buffers for code and data cache blocks. We are currently investigating the performance benefit of these different design aspects.

Another possible use of hardware page placement is dynamic load balancing cache usage between multiple threads or processors sharing an L2 cache. By placing pages in certain locations in the L2 cache we can restrict processes to a fixed area in the cache. The area can be expanded dynamically to fit the individual working set sizes and cache usage, and processes with non-conflicting working sets can be mapped to the same area in the cache.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. In addition, we would like to thank Amitabh Srivistava and Alan Eustace for providing ATOM, and Todd Austin and Doug Burger for providing SimpleScalar. This work was funded in part by NSF CAREER grant No. CCR-9733278, a gift from Microsoft, and a grant from Compaq Computer Corporation.

References

[1] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):179–190, May 1993.

[2] B. Bershad, D. Lee, T. Romer, and J.B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.

[3] E. Bugnion, J. Anderson, T. Mowry, M. RosenBlum, and M. Lam. Compiler-directed page coloring for multiprocessors. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[4] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[5] B. Calder, D. Grunwald, and J. Emer. Predictive sequential associative cache. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, February 1996.

[6] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[7] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 28(5):252–262, October 1994.

[8] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, January 1999.

[9] T.M. Conte, K.N. Menezes, P.M. Mills, and B.A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

[10] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.

[11] K. Ghose and M.B. Kamble. Energy efficient cache organizations for superscalar processors. In *Power-Driven Microarchitecture Workshop*, June 1998.

[12] N. Gloy, T. Blockwell, M.D. Smith, and B. Calder. Procedure placement using temporal ordering information. In *30th International Symposium on Microarchitecture*, December 1997.

[13] L. Hammond, B. Nayfeh, and K. Olukotun. A single-chip multiprocssor. *IEEE Computer, Special Issue on Billion-Transistor Processors*, September 1997.

[14] K. Harty and D.R. Cheriton. Application-controlled physical memory using external page cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, October 1992.

[15] A.H. Hashemi, D.R. Kaeli, and B. Calder. Efficient procedure mapping using cache lince coloring. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 171–182, June 1997.

[16] W.W. Hwu and P.P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual International Symposium on Computer Architecture*, pages 242–251. ACM, 1989.

[17] B. Jacob and T. Mudge. A look at several memory management units, tlb-refill mechanisms,and page table organizations. In *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[18] T. Johnson, M. Merten, and W. Hwu. Run-time spatial locality detection and optimization. In *30th International Symposium on Microarchitecture*, December 1997.

[19] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[20] M.B. Kamble and K. Ghose. Analytical energy dissipation models for low-power caches. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, pages 143–148, 1997.

[21] R. Kessler and M. Hill. Page placement algorithms for large real-indexed caches. *Transactions on Computer Systems*, 10(4), November 1992.

[22] R. E. Kessler. *Analysis of Multi-Megabyte Secondary CPU Cache Memories*. TR 1032, Computer Sciences Department, UW–Madison, Madison, WI, July 1991.

[23] R. E. Kessler, R. Jooss, A. Lebeck, and M. D. Hill. Inexpensive implementations of set-associativity. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 17(3):131–139, 1989.

[24] S. McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 183–191, April 1989.

[25] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *Transactions on Programming Languages and Systems*, 18(4), July 1996.

[26] P. Panda, N. Dutt, and A. Nicolau. Memory data organization for improved cache performance in embedded processor applications. *Transactions on Design Automation of Electronic Systems*, 2(4), October 1997.

[27] J. Peir, Y. Lee, and W. Hsu. Capturing dynamic memory reference behavior with adaptive cache topology. In *Eigth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[28] K. Pettis and R. C. Hansen. Profile guided code positioning. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, 25(6):16–27, June 1990.

[29] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, June 1998.

[30] T. Romer, D. Lee, B. Bershad, and J.B Chen. Dynamic page mapping policies for cache conflict resolution on standard hardware. In *Proceedings of the 1st Symposium on Operatingy Systems Design and Implemenation*, pages 255–266, November 1994.

[31] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[32] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[33] G. Tyson and M. Farrens. Managing data caches using selective cache line replacement. *International Journal of Parallel Programming*, 25(3), June 1997.

[34] S. J.E. Wilton and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. Tech report 93/5, DEC Western Research Lab, 1994.

[35] Y. Yamada, J. Gyllenhaal, G.Haab, and W. W. Hwu. Data relocation and prefetching for large data sets. In *27th International Symposium on Microarchitecture*, pages 118–127, December 1994.