
WRL

Research Report 96/3



Efficient Procedure Mapping using Cache Line Coloring

Amir H. Hashemi
David R. Kaeli
Brad Calder

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and technical notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	JOVE::WRL-TECHREPORTS
Internet:	WRL-Techreports@decwrl.pa.dec.com
UUCP:	decpa!wrl-techreports

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word "help" in the Subject line; you will receive detailed instructions.

Reports and technical notes may also be accessed via the World Wide Web:
<http://www.research.digital.com/wrl/home.html>.

**Efficient Procedure
Mapping using
Cache Line Coloring**

**Amir H. Hashemi
David R. Kaeli
Brad Calder**

October 1996



Abstract

As the gap between memory and processor performance continues to widen, it becomes increasingly important to exploit cache memory effectively. Both hardware and software approaches can be explored to optimize cache performance. Hardware designers focus on cache organization issues, including replacement policy, associativity, block size and the resulting cache access time. Software writers use various optimization techniques, including software prefetching, data scheduling and code reordering. Our focus is on improving memory usage through code reordering compiler techniques.

In this paper we present a link-time procedure mapping algorithm which can significantly improve the effectiveness of the instruction cache. Our algorithm produces an improved program layout by performing a color mapping of procedures to cache lines, taking into consideration the procedure size, cache size, cache line size, and call graph. We use cache line coloring to guide the procedure mapping, indicating which cache lines to avoid when placing a procedure in the program layout. Our algorithm reduces on average the instruction cache miss rate by 45% over the original mapping and by 14% over the mapping algorithm of Pettis and Hansen.

1 Introduction

The increasing gap between processor and main memory speeds has forced computer designers to exploit cache memories. A cache is smaller than the main memory and, if properly managed, can hold a major part of the working set of a program [6]. The goal of memory subsystem designers is to improve the average memory access time. Reducing the cache miss rate is one factor for improving memory access performance. Cache misses occur for a number of reasons: cold start, capacity, and collisions [12]. A number of cache line replacement algorithms have been proposed to reduce the number of cache collisions [2, 13, 17].

Instead of concentrating on cache organization we concentrate on the layout of a program on the memory space. Bershad et.al. suggested remapping cache addresses dynamically to avoid conflict misses in large direct-mapped caches [3]. An alternative approach is to perform code repositioning at compile or link-time [8, 10, 11, 15, 18]. The idea is to place frequently used sections of a program next to each other in the address space, thereby reducing the chances of cache conflicts while increasing spatial locality within the program.

Code reordering algorithms for improved memory performance can span several different levels of granularity, from basic blocks, to loops, and to procedures. Research has shown that basic block reordering and procedure reordering can significantly improve a program's execution performance. Pettis and Hansen [11] found that the reduction in execution time when using procedure reordering was around 8%, and the reduction in execution time for basic block reordering was around 12% on an HP-UX 825 architecture with a 16K direct mapped unified cache. When both of the optimizations were applied together an average improvement of 15% was achieved.

The mapping algorithm we propose in this paper improves upon prior work, particularly when a program's control flow graph is larger than the cache capacity. Since we are interested in dealing with graphs that are larger than the target instruction cache, we concentrate our discussion in this paper on reordering procedures. Even so, our algorithm can also be used with, and can benefit from, basic block reordering and procedure splitting, as described later in §5.

Our research differs from prior research in procedure reordering because our algorithm uses the cache size, cache line size, and the procedure size to perform a color mapping of cache lines to procedures. This color mapping allows our algorithm to intelligently place procedures in the layout by preserving color dependencies with a procedure's parents and children in the call graph, resulting in fewer instruction cache conflicts.

In this paper we will describe our algorithm and demonstrate its merit through trace-driven cache simulation. In §2 we describe our color mapping algorithm and compare our algorithm with prior work

⁰Authors' addresses: A.H. Hashemi and D.R. Kaeli are at the Department of Electrical and Computer Engineering, Northeastern University, 318 Dana Research Center, Boston, MA 02115; B. Calder is at the Department of Computer Science and Engineering 0114, University of California, San Diego, 9500 Gilman Road, La Jolla CA 92093-0114.

in code reordering. The methodology used to gather our results is described in §3. In §4 we provide quantitative results using our improved procedure ordering algorithm. We then discuss implications and future work for our algorithm in §5, and we summarize our contributions in §6.

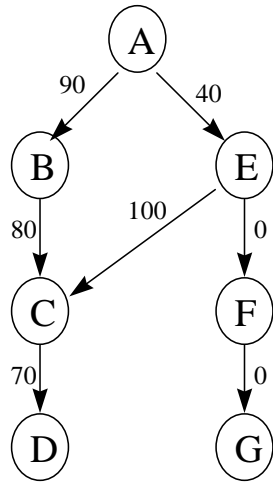
2 Procedure Mapping

In this section we describe our procedure mapping algorithm. For the following description, we will assume that the instruction cache is direct mapped (in §5 we discuss how to apply our algorithm to set-associative caches). The basic idea behind the algorithm is to treat the memory address space as two dimensional by breaking up the address space into pieces that are equivalent to the size of the cache, and using the cache blocks occupied by each procedure to guide the mapping. In contrast, previous research has treated memory layout as a one dimensional address space. Employing a second dimension allows our algorithm to intelligently avoid cache conflicts when mapping a procedure for the first time, and it provides the ability to move procedures that have already been mapped in order to eliminate additional conflicts as they arise. To avoid conflicts, we keep track of the colors each procedure is mapped to and a set of colors indicating which colors are currently unavailable to that procedure. We will refer to this set of colors as the *unavailable-set*.

For a given procedure, the unavailable-set of colors represents the colors occupied (i.e., cache lines used) by all of the immediate parents and children of that procedure in the call graph which have already been mapped to cache lines. Our algorithm uses a call graph with weighted procedure call edges for indicating the importance of mapping procedures next to each other. The algorithm concentrates on only eliminating *first-generation* cache conflicts, which are the conflicts between a procedure and the immediate parents and children of that procedure in the call graph. When mapping a procedure, our algorithm tries to avoid cache conflicts by avoiding cache line colors in its unavailable-set. Once a procedure has been mapped, a procedure can later be moved to a new location without causing cache conflicts, as long as it does not move to a location (color) which is in its unavailable-set. In using the color mapping to place and move procedures in this way, we are guaranteed that the new location will not increase the number of first-generation conflicts for the procedures in our call graph.

One of the hurdles in a mapping algorithm where code is allowed to move after it has been already mapped, is the problem of how to handle the empty space left behind by the moved procedures. If possible, this gap should be filled since the program is laid out in a contiguous memory space. Therefore, moving a procedure should be followed by filling the space left by the procedure with other procedures, otherwise this can result in a chain of relocations that are hard to manage.

Studies of program behaviors show that 10% to 30% of a program accounts for 90% of its execution time [5]. The rest of the code is not heavily exercised or is often not even executed. Our algorithm takes advantage of this property by dividing each program into frequently executed (*popular*) and infrequently



Procedure	# Cache Lines
A	1
B	1
C	2
D	2
E	2
F	1
G	2

Figure 1: Example call graph. Each node represents a procedure and each edge represents a procedure call. The numbers associated with each edge indicates the number of times the procedure call was executed. The table shows for each procedure how many cache lines is needed to hold the procedure.

executed (*unpopular*) procedures. The unpopular procedures are treated as fluff or glue, and are used to fill the empty space left behind by moved procedures in our algorithm. We will not worry about conflicts when positioning unpopular procedures, since these parts of a program do not significantly contribute to the number of first level cache conflicts.

2.1 Cache Coloring Algorithm

We will now describe the details of our block coloring algorithm and use an example to demonstrate how to layout procedures. Figure 1 presents an example call graph, containing 7 procedures *A* through *G*, where nodes represent procedures and the edges represent procedure calls. Each edge contains a weight indicating how many times that procedure was called. The Figure also contains a table indicating the number of cache blocks each procedure occupies. In this example and algorithm description, we assume the instruction cache is direct mapped and contains only 4 cache lines.

Figure 2 shows the steps taken by our algorithm in mapping the example call graph given in Figure 1. The cache is divided into a set of colors, one color for each cache block. The four cache lines are given the colors *red*, *green*, *blue*, and *yellow*. In Figure 2, the first column shows at each step which edge or procedure is being processed. The second column shows which of the four edge processing cases the current step corresponds to in our algorithm. The third column shows the current mapping of the processed procedures and edges over the colored 4 block cache space. The last column shows the changes to the unavailable-set of colors for the procedures being processed at each step. If a procedure spans multiple

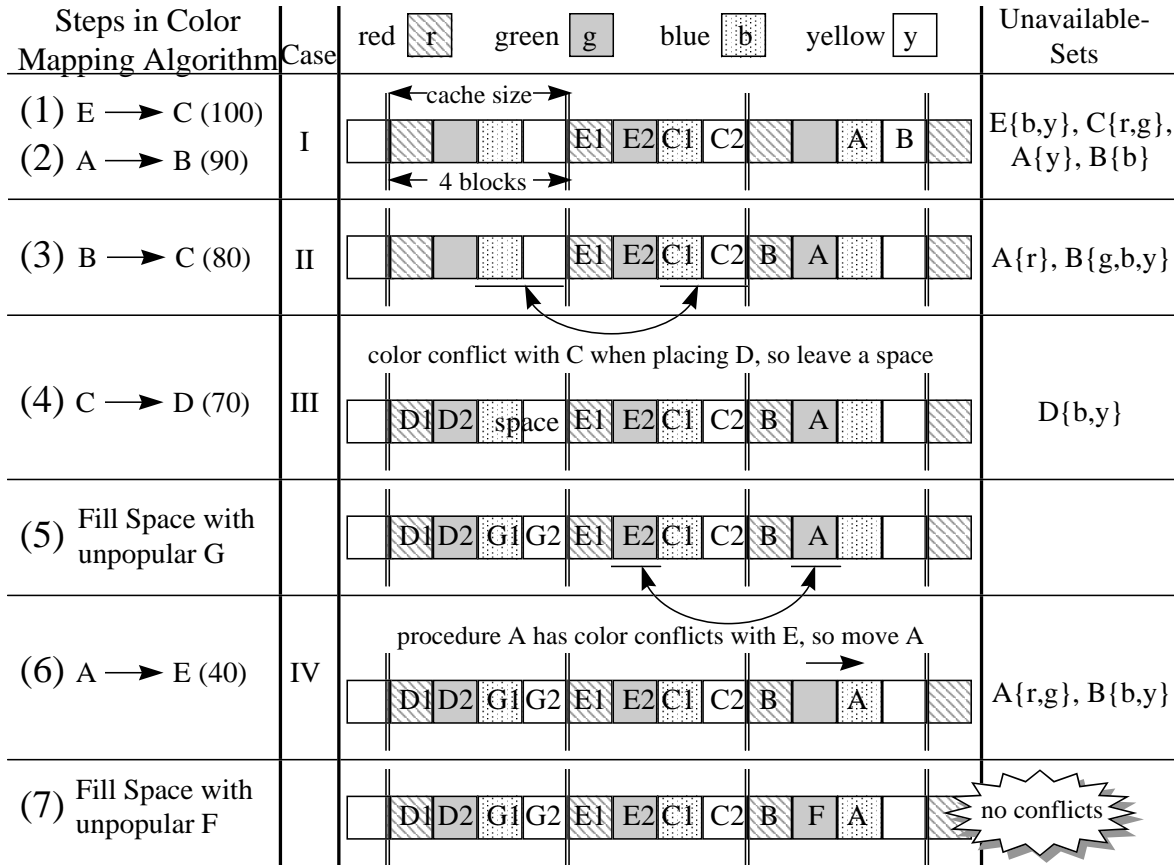


Figure 2: Procedure mapping using cache line coloring. The first column indicates the steps taken in our color mapping algorithm and each edge and procedure processed at each step. The second column shows which of the four edge processing cases the current step corresponds to in our algorithm. The third column shows the address space divided into sizes equal to the instruction cache, and shows the mapping of the program at each step. The instruction cache contains 4 blocks labeled: *red*, *green*, *blue*, and *yellow*. The last column shows the unavailable-sets as they are changed for the procedures at each step in the algorithm.

cache lines (as does C in our example), it will generate multiple mappable elements (e.g., $C1$ and $C2$), as is shown in Figure 2.

Our algorithm maintains three important pieces of state for each procedure: the number of cache lines (colors) needed to hold the procedure, the cache colors used to map the procedure, and the unavailable-set of colors which represents the cache lines where the procedure should not be mapped to. We do not actually store the unavailable-set of colors. Instead, each procedure contains pointers to its parents and children in the call graph. The unavailable-set of colors is then constructed for a procedure as needed by unioning all the colors used to map each of the procedure's parents and children, *only* if the edge joining the procedure to the parent or child has already been processed in the algorithm.

Our algorithm starts by building a procedure call graph, similar to the one shown in Figure 1. Every procedure in the program is represented by a node in the graph, and each edge between nodes represents a procedure call. Multiple call sites to the same procedure from a single procedure are represented as a single edge in our call graph. The edge values represent the number of times each edge (i.e., call path) was traversed. The sum of the edge weights entering and exiting a node indicates the number of incoming and outgoing procedure calls and this determines that procedure's popularity.

After the call graph is built, the popularity of each procedure is considered. Based on popularity, the graph is split into the *popular* procedures and edges and the *unpopular* procedures and edges. The popular procedure set contains those procedures which are frequently a caller or a callee, and the popular edge set contains the frequently executed procedure call edges. The unpopular procedures and edges are those not included in the above two popular sets. Note, there is a difference between popular procedures and time consuming procedures (procedures that consume a noticeable portion of a program's overall execution time). A time consuming procedure may be labeled unpopular because it rarely switches control flow to another procedure. If a procedure rarely switches control flow, one does not have to worry about eliminating cache conflicts between this procedure and the rest of the call graph. In the example in Figure 1, popular procedures are A , B , C , D , and E , and the unpopular procedures are F and G since they are never executed. The popular edges are $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, $A \rightarrow E$, and $E \rightarrow C$, and the unpopular edges are $E \rightarrow F$ and $F \rightarrow G$. The algorithm then sorts the popular edges in descending order using the edge weights. The unpopular procedures are sorted by procedure size, and are used to fill in spaces created by our color mapping.

After the program's popularity has been decided, we process all of the popular edges starting with the most frequently executed and ending with the least frequently executed. There are four possible cases when processing an edge in our algorithm. The first case occurs when an edge connects two procedures that have not yet been mapped. In this case, the two procedures are merged into a *compound* node. The two procedures are placed next to each other in the layout and they are assigned cache line colors starting at an arbitrary color (position). Each procedure is assigned the number of cache line colors equal

to $(\text{procedure's size in bytes})/(\text{cache line size in bytes})$. After the colors have been assigned, the unavailable-set for each procedure includes the colors (cache lines) used by the other procedure at the other end of the call edge. The remaining three cases encountered when processing an edge include: when the call edge links two procedures in two different compound nodes, when the edge is between an unprocessed procedure and a procedure in a compound node, and when the edge being processed is a call between two procedures in the same compound node. The following four paragraphs discuss the details for the four edge processing cases in our algorithm.

Case I: The first case, when an edge connects two unmapped procedures, is shown in the first two steps of Figure 2. The algorithm starts with the heaviest edge (most heavily traversed) in the call graph's set of popular edges, $E \rightarrow C$, and forms a compound node $E - C$. This compound node is arbitrarily mapped to the cache line colors. The unavailable-set of colors for E now includes *blue* and *yellow* (the colors C maps to) and the unavailable-set for C now includes *red* and *green* (the colors E maps to). The second step in Figure 2 processes the edge $A \rightarrow B$ between two unmapped procedures. The two procedures are combined into a compound node, and their unavailable-sets are shown in the Figure. Note that the unavailable-set for A does not include colors *red* and *green*, even though there is an edge $A \rightarrow E$ in the call graph and node E is mapped to the colors *red* and *green*. This is because the procedure's unavailable-set only includes parent and children procedures connected by edges that have been processed, and the edge $A \rightarrow E$ has not yet been processed. We chose this restriction since the unavailable-set of colors is used to restrict where to place procedures, and when placing a procedure, the procedure should only be restricted by the edges with the heaviest (most important) weights.

Case II: The second case occurs when the edge being processed connects two procedures in different compound nodes. For this case, the two compound nodes are merged together, concatenating the compound node that is shorter in length (number of procedures) to the larger compound node. This is shown in step 3 of Figure 2 for edge $B \rightarrow C$, which combines two compound nodes $E - C$ and $A - B$. The compound nodes both contain the same number of procedures, so we arbitrarily choose $A - B$ to be the smaller compound node. Our algorithm now decides where to map, and how to order, $A - B$ since there are four possibilities: $A - B - E - C$, $B - A - E - C$, $E - C - A - B$ and $E - C - B - A$. The first decision to make is on which side of compound node $E - C$ should $A - B$ be placed. This is decided by taking the shortest $\text{mod}(\text{distance to procedure in compound node}/\text{cache size})$. For our example, the distance to C is used and is calculated to be the distance in the number of cache line colors from the middle of procedure C to each end of the compound node. From the mapping in step 1 of Figure 2, this distance is 1 cache line to the right of C in the compound node $E - C$ and 3 cache lines to the left of C in compound node $E - C$. Therefore the algorithm decides to place $A - B$ to the right of $E - C$. The $\text{mod}(\text{distance to procedure}/\text{cache size})$

heuristic is used to increase the probability of being able to easily map the 2nd compound node to non-conflicting cache colors. Note, that placing $A - B$ to the right of $E - C$ produces a mapping where no cache conflicts occur, whereas if we would had chosen to put $A - B$ on the left side of $E - C$ this would have caused a cache coloring conflict. The next decision our algorithm makes is in which order to place $A - B$, either $E - C - A - B$ or $E - C - B - A$. This is decided by choosing the ordering so the two procedures connected by the edge being processed (i.e., $B \rightarrow C$) are closest to each other in the program layout. Thus we arrive at a mapping of $E - C - B - A$. After this is decided, the algorithm makes sure that the two nodes for the edge being processed, B and C , have no cache lines that conflict. This is done by comparing the colors used by C with the colors used by B . If there is a conflict, the smaller compound node is shifted away from the larger compound node until there is no longer a conflict. The space left in the mapping will be filled with unpopular procedures. If a conflict cannot be avoided then the original location is used. When the final position for the smaller compound node is determined, the algorithm goes through each procedure and updates the colors (cache lines) used by each procedure. Notice that this changes the unavailable-set of colors: A 's set of unavailable colors changes to *red* and B 's changes to *green*, *blue* and *yellow*.

Case III: The third type of edge connects an unmapped procedure and a procedure in a compound node. We process this case similarly to case II as described in the previous paragraph. In this situation, the unmapped procedure is placed on either end of the compound node, which side is decided by using the shortest $\text{mod}(\text{distance to procedure}/\text{cache size})$ heuristic as described above. Once a side is chosen, the cache line colors used by the newly mapped procedure are checked against the colors used by its corresponding procedure in the compound node. If there is a conflict, space is inserted in the address space between the newly mapped procedure and the compound node until the newly mapped procedure can be assigned colors which do not conflict. If this is not possible, the procedure is left at its original position, adjacent to the compound node. Step 4 in Figure 2 shows this scenario. The algorithm next processes edge $C \rightarrow D$, where C is contained in a compound node and D has not yet been mapped. The algorithm first decides on which side of the compound node to place D . Since both of the distances to the middle of C are the same (3 cache lines), the algorithm arbitrarily chooses a side and D is placed to the left of the compound node. The colors used for D at this location are *blue* and *yellow*. This would create a conflict since those colors overlap with the colors used by C . Therefore the algorithm shifts D to the left until it finds a suitable location (if possible) where D no longer conflicts with C . This location for D is found at the colors *red* and *green*. This leaves a space in the compound node, as shown in step 4. If a space is created inside of a compound node, the space is filled with the largest unpopular procedure which will fit. This is shown in step 5 of Figure 2, where the space created by shifting D is filled with the unpopular procedure G .

Case IV: The fourth and final case to handle occurs when the edge being processed has both procedures belonging to the same compound node. This is a very important case since the algorithm finally gets to use the unavailable-set to avoid cache conflicts. If the colors used by the two procedures of the edge overlap (conflict), then the procedure closest (in terms of cache lines) to either end of the compound node is moved past the end of the compound node, creating a space or gap in the compound node where it use to be located. This space will later be filled by an unpopular procedure or procedures. The unavailable-set for the procedure that is moved past the end of the compound node is updated to include the colors of the corresponding procedure left inside the compound node. The algorithm then checks to see if the current colors used by the procedure conflict with any of its unavailable colors. If there is a conflict, the procedure is shifted away from the compound node in the address space until there is no longer a conflict with its unavailable-set of colors. If we are unable to find a non-conflicting location for the procedure, the original location inside the compound node is used. This final scenario is shown in step 6 in Figure 2, where the edge from $A \rightarrow E$ is processed and its two procedures are in the same compound node. In examining the colors used by both A and E , we see that the two procedures' colors conflict since they map to the same cache block (*green*). The algorithm tries to eliminate this conflict by choosing to move A , since it is the closest to an end of the compound node. The algorithm moves A past the end of the compound node, mapping it to the color *blue*. When checking A 's new mapping against its unavailable-set (*red* and *green*), no conflicts are found, so this is an acceptable location for procedure A . Using the unavailable-set in this way guarantees that previous mappings for A take precedence over the edge $A \rightarrow E$, because those mappings were more important. Finally, since A was moved in step 6, it created a space in the compound node, as shown in Figure 2. After any space is made inside of a compound node, that gap is filled with a procedure(s) from the unpopular list. In our example, the remaining procedure F is used to fill the gap. We then arrive at the final mapping as shown in step 7, which has no first-generation cache conflicts.

This process is repeated, until all of the edges in the popular set have been processed. Any remaining procedures in the unpopular list are mapped using a simple depth-first traversal of the unpopular edges that join these unpopular procedures. This can create several disjoint compound nodes. These nodes are then ordered in the final layout, from the most frequently executed to the least frequently executed.

2.2 Comparison to Previous Work

There has been considerable work in the area of profile-driven program optimizations and procedure reordering. We now discuss relevant previous work and how it relates to our algorithm.

2.2.1 Knowledge of Cache Size

McFarling examined improving instruction cache performance by not caching infrequently used instructions and by performing code reordering compiler optimizations [10]. The mapping algorithm works at the basic block level and concentrates on laying out the code based on loop structures in the program. The algorithm constructs a control flow graph with basic block, procedure, and loop nodes. It then tries to partition the graph, concentrating on the loop nodes, so that the height of each partitioned tree is less than the size of the cache. If this is the case, then all of the nodes inside of the tree can be trivially mapped since they will not interfere with each other in the cache. If this is not the case, then some nodes in the mapping might conflict with others in the cache.

The notion of wanting the mapped tree size smaller than the cache size also applies to our algorithm when we partition the call graph into popular and unpopular procedures and edges. Partitioning the call graph actually splits the graph into several disjoint subgraphs comprised of the popular procedures and edges. This has the effect of breaking the call graph into smaller, and more manageable, pieces. If the sum of all the procedure sizes in a subgraph is smaller than the size of the instruction cache, then there will be no conflicting colors when laying out all of the procedures in the subgraph and the mapping can be done trivially as suggested by McFarling. The benefit of our algorithm over McFarling's is that instead of just taking into consideration the cache size we also take into consideration the exact cache lines used by each procedure in the mapping. This allows our algorithm to effectively eliminate first-generation cache conflicts, even when the popular subgraph size is larger than the instruction cache, by using the color mapping and the unavailable-set of colors.

Torrellas, Xia and Daigle [18] (TXD) also described an algorithm for code layout for operating system intensive workloads. Their work takes into consideration the size of the cache and the popularity of code. Their algorithm partitions the operating system code into executed and non-executed parts at the basic block level. It then repeatedly creates sequences of basic blocks from the executed code. All the basic blocks with weights above a threshold value are removed from the graph and put into a *sequence*, which is a list of basic blocks. All the basic blocks in a sequence are then layed out together in the address space. The threshold value is then lowered and the process is repeated until all the frequently executed basic blocks have been put into sequences. Their algorithm takes into consideration the cache size by mapping the most frequently executed sequence into a special area in the cache. The rest of the sequences are then mapped to areas in the cache, avoiding this special area. This creates gaps in the program layout which are then filled by the non-executed basic blocks. The TXD algorithm is designed for mapping operating system code to increase performance, by keeping commonly used system code in the cache. Our algorithm is designed for application code and tries to eliminate as many first-generation conflicts as possible. These two goals are different and may champion the use of different algorithms. The techniques used by TXD, which work well

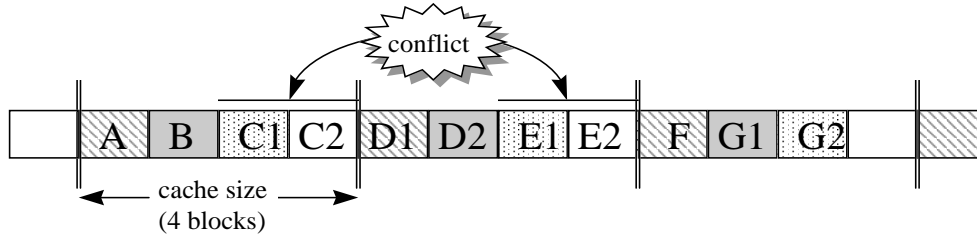


Figure 3: Procedure mapping for a greedy depth-first traversal of the call graph.

for operating system code, may not work as well to eliminate first-generation cache conflicts in application code.

As described in §2.1, our algorithm uses unpopular procedures in a manner similar to how TXD uses non-executed operating system basic blocks. We use the unpopular code in an application to fill in spaces created when mapping procedures. The two approaches differ in that our algorithm uses the unpopular procedures to try to eliminate cache conflicts for *all* popular procedures by performing a color mapping that gives priority to the procedures that switch control flow the most in the call graph. In comparison, TXD uses the non-executed code to eliminate cache conflicts for only some of the popular basic blocks: the most frequently executed sequence(s). Keeping track of the colors used by each procedure, and using the unavailable-set to eliminate as many conflicts as possible, makes our algorithm more general for eliminating first-generation conflicts in application code.

Another technique used by TXD which works well for operating system code, but which may not work as well for application code, is recursively breaking up the basic blocks into sequences using a threshold value. This technique does not take into consideration the connectivity of the basic blocks in the sequence. Therefore a sequence could be layed out together in the address space, with the basic blocks having little or no temporal locality, and the basic blocks in one sequence could cause conflict misses with basic blocks in another sequence. For application code, our coloring algorithm offers better performance over a recursive threshold partitioning algorithm since we take into consideration the connectivity of the graph.

2.2.2 Procedure Mapping

Hwu and Chang described an algorithm for improving instruction cache performance using inlining, basic block reordering, and procedure reordering compiler optimizations [8]. Their algorithm builds a call graph with weighted call edges produced by profiling. For the procedure reordering, their algorithm processes the call graph depth first, mapping the procedures to the address space in depth first order. Their depth-first traversal is guided by the edge weights determined by the profile, where a heavier edge is traversed (layed

Important points of decision in the Pettis and Hansen algorithm

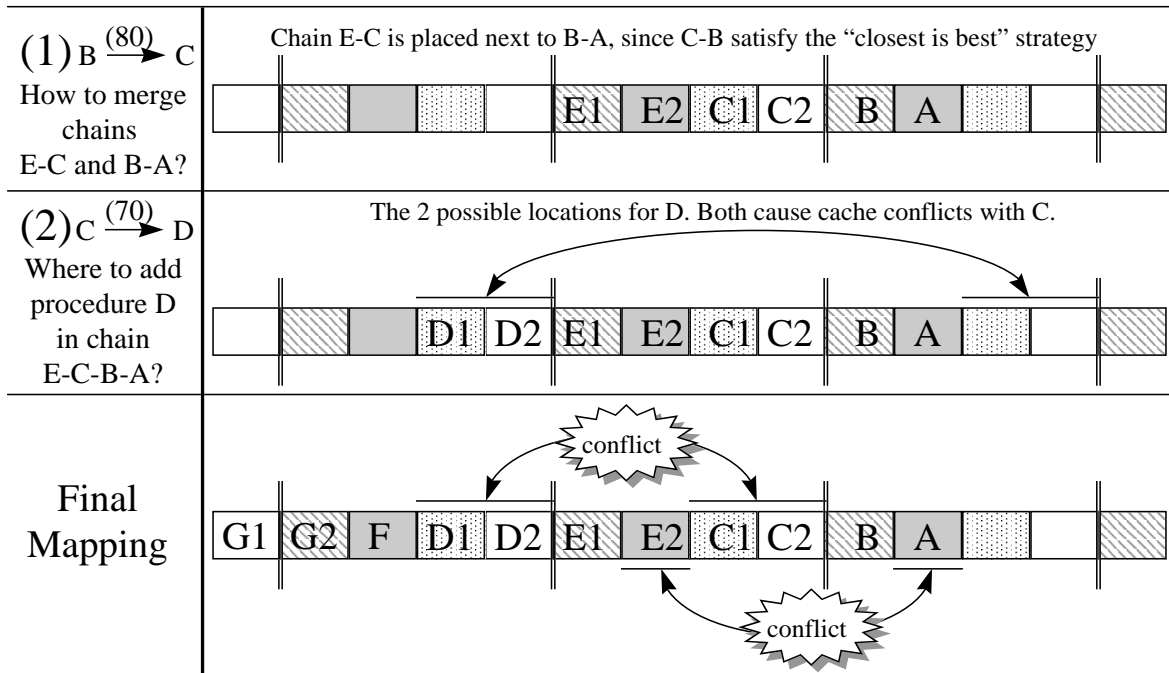


Figure 4: Procedure mapping for the Pettis and Hansen greedy algorithm.

out) before an infrequently executed edge. In using the call graph shown in Figure 1, a depth-first traversal following the most frequently executed edges would traverse the edges in order of $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow D$, $A \rightarrow E$, $E \rightarrow C$, $E \rightarrow F$, and $F \rightarrow G$. Figure 3 represents the final mapping achieved by their algorithm. The drawback of this approach occurs when the depth-first traversal follows an unimportant path in the control flow graph, which will then lay out unpopular procedures before considering procedures on a more important path. This is seen in Figure 1 where their algorithm processes the edge $C \rightarrow D$ before the edge $E \rightarrow C$. This can create significant first-generation cache conflicts in the call graph, as seen by the conflict between procedures E and C in Figure 3.

Pettis and Hansen [11] also described a number of techniques for improving code layout that include: basic block reordering, procedure splitting, and procedure reordering. Their algorithm employs a closest-is-best strategy to perform procedure reordering. The reordering starts with the heaviest executed call edge in the program call graph. The two nodes connected by the heaviest edge will be placed next to each other in the final link order. This is taken care of by merging the two nodes into a *chain*. The remaining edges entering and exiting the chain node are coalesced. This process continues until the whole call graph is merged into chains which can no longer be merged. Figure 4 shows the key points of the Pettis and

Hansen [11] procedure mapping algorithm when processing the call graph in Figure 1. Their algorithm starts by processing edge $E \rightarrow C$, merging nodes E and C into a chain $E - C$. This is followed by edge $A \rightarrow B$, where A and B are merged into a chain $A - B$. The next edge to be processed is $B \rightarrow C$. This brings the algorithm to the first point shown in Figure 4, which is how to merge the chains $E - C$ and $A - B$. At this point their algorithm uses a *closest-is-best* heuristic, and chooses to place procedure B next to C , since the edge $B \rightarrow C$ has a stronger weight than $A \rightarrow E$. The next edge to be processed is from $C \rightarrow D$. This means procedure D needs to be placed at the front or end of chain $E - C - B - A$. Figure 4 shows that, no matter which side of the chain D is placed, a first-generation cache conflict will occur with C . This illustrates the main drawback of their approach, which is that the algorithm fails to monitor the chain size. Therefore, once a chain becomes larger than the size of the instruction cache, the effectiveness of their closest-is-best strategy and node merging strategy, decreases. In looking at the final mapping in Figure 4, we see that the mapping has first-generation conflicts between procedures A and E , and procedures C and D .

Our algorithm improves on the Hwu and Chang and the Pettis and Hansen procedure reordering algorithms by keeping track of the cache lines (colors) used by each mapped procedure when performing the procedure mapping. This allows us to effectively map procedures, eliminating cache conflicts even when the compound node size grows larger than the instruction cache. Neither of their algorithms take into consideration the attributes of the cache, such as cache size, line size, and associativity. They also do not consider leaving spaces in their layout, which can be used to reduce the number of cache conflicts. As shown in Figure 2, when using our color mapping algorithm, no first-generation cache conflicts occur for the call graph shown in Figure 1. In comparison, Figure 3 and Figure 4 show that both the Hwu and Chang and the Pettis and Hansen algorithms suffer from first-generation cache conflicts for the reasons discussed above.

3 Methodology

To evaluate the performance of our algorithm, we modified `gcc` version 2.7.2 to use our new procedure mapping algorithm when linking an application. This has restricted the type of applications we can examine in this study to programs that can be compiled with `gcc`. Therefore, the programs we examined are from the SPECInt95 suite, SPECInt92 suite, and three `gnu` applications.

We used trace driven simulation to quantify the instruction cache performance of our algorithm [9]. The trace driven simulations were obtained using `ATOM`, an execution-driven simulation tool available from Digital Equipment Corporation [16]. `ATOM` allows instrumentation of binaries on DEC Alpha processors and can produce the necessary information about the frequency of procedure calls, procedure sizes, and the program's control flow graph. In our simulations we model a direct-mapped 8 kilobyte instruction cache

Program	Input	# Instrs Traced in Millions	Exe Size K-Bytes	# Static Procs	Popular Procedures		Unpopular Filler % Exe Size
					% Exe Size	% Procs	
li	li_input	6938 M	417 K	575	6% (24 K)	15%	0.5% (2 K)
m88ksim	dcrand.lit	27942 M	557 K	460	7% (41 K)	9%	0.9% (5 K)
perl	scrabbl	592 M	819 K	557	4% (20 K)	4%	0.6% (5 K)
espresso	tial	1145 M	516 K	539	12% (60 K)	17%	0.5% (3 K)
eqntott	int_pri_3	2021 M	400 K	498	11% (46 K)	7%	0.8% (3 K)
bison	objc_parse	77 M	352 K	369	9% (32 K)	10%	1.0% (4 K)
flex	fixit.l	24 M	492 K	668	11% (52 K)	8%	0.8% (4 K)
gzip	gcc-2.7.2.tar	9242 M	344 K	140	3% (10 K)	21%	0% (0 K)

Table 1: Measured attributes of traced programs. The input is used to both profile the program and gather performance results. The attributes include the number of instructions traced when simulating the program, the executable size of the program, and the number of static procedures in the program. Also shown is the percentage of the executable and the percentage of static procedures that the popular procedures account for after partitioning the program into popular and unpopular procedures when using the color mapping algorithm. The last column shows the percentage of unpopular procedures in terms of the size of the executable that were used as fluff (to fill in spaces) in our color mapping algorithm.

with a 32 byte block size, similar to the size used for the DEC Alpha 21064 and DEC Alpha 21164 first-level instruction cache. Therefore, in our color mapping, the number of colors is equal to 256, which is equal to the number of direct mapped cache blocks.

Table 1 describes the static and dynamic attributes for the programs we studied. The first column contains the program name, and the second column shows the input used to profile each program. The third column shows the number of instructions traced for the input used. The fourth column shows the size of each program in kilobytes, and the fifth column shows the number of static procedures in the program. The next two columns show results for the popular procedures in the program as determined by our color mapping algorithm described in §2.1. The sixth column shows the percentage of the executable that contains only the popular procedures, and the seventh column shows the percentage of static procedures which are considered popular. The final column shows the percentage of the executable which were unpopular procedures used as filler to fill in spaces created in the color mapping (as described in §2.1). We used profile information to guide the partitioning of the program into popular and unpopular parts. All the procedures and edges that account for less than 1% of the switches in control flow in the call graph are labeled as unpopular. We can see that by splitting each program into popular and unpopular sets, that the popular procedures make up only 3% to 12% of the static executable size, and this accounts for 4% to 21% of the static procedures in the program. Mapping these procedures correctly will eliminate most of the cache conflicts in the application for the inputs we examined.

Program	I-Cache Miss Rate			Miss Rate Reduction Over		# Instruction Cache Misses		
	Original	P&H	Color	Original	P&H	Original	P&H	Color
li	1.4%	0.3%	0.3%	79%	0%	97,127,676	23,786,704	19,943,570
m88ksim	3.0%	1.7%	1.4%	53%	18%	838,249,456	475,008,025	391,183,079
perl	7.5%	4.7%	4.4%	41%	6%	44,378,149	27,810,306	26,035,180
espresso	0.9%	0.9%	0.5%	44%	44%	10,308,276	10,211,819	6,435,699
eqntott	0.2%	0.3%	0.1%	50%	66%	4,042,293	6,741,562	2,730,813
bison	1.5%	1.5%	1.1%	27%	27%	1,153,805	1,132,929	842,057
flex	2.2%	1.7%	1.7%	23%	0%	525,433	407,423	399,784
gzip	1.1%	0.0%	0.0%	99%	0%	101,667,137	33,950	29,370
Average	2.2%	1.4%	1.2%					

Table 2: Instruction cache performance for the Original mapping, Pettis and Hansen (P&H) mapping, and our Color mapping algorithm. The first three column shows the instruction cache miss rates. The next two columns show the percent reduction in the miss rates when using our Color mapping algorithm in comparison to the Original and P&H procedure mapping. The last three columns show the number of instruction cache misses.

4 Results

To evaluate the performance of our color mapping algorithm we also implemented the Pettis and Hansen algorithm described in Section §2.2. Table 2 shows the instruction cache miss rates for the original program, the Pettis and Hansen algorithm, and our cache coloring algorithm. For the results shown, the same input used in Table 1 was used for both profiling the program and gathering the results. The second column provides the cache miss ratio for the *Original* program using the standard link order for the benchmark executables as specified in the makefile provided with the programs. The next column indicates the cache miss ratio after applying the Pettis and Hansen (*P&H*) algorithm. The fourth column, labeled *Color*, refers to the new link order produced by our cache color mapping algorithm. The next two columns show the percent reduction in the cache miss rate when using our algorithm in comparison to the original program and the P&H mapping. The last three columns show the number of instruction cache misses for the original program, P&H layout, and our color mapping.¹

As seen in Table 2, when using the color mapping algorithm the miss rate of the original program is decreased on average by 45%, with reductions as high as 99% for *gzip*. In comparison to the P&H algorithm our color mapping reduces the miss rate on average by 14%. The Table shows that in comparison to P&H our algorithm provides a substantial reduction in the cache miss rate for the 4 programs *m88ksim*, *espresso*, *eqntott*, and *bison*, provides a smaller improvement for *perl*, and has approximately the same instruction cache miss ratio for *li*, *flex*, and *gzip*.

¹Only averages are shown for the miss rate columns, since the averages for the other columns in the table are not meaningful.

Our algorithm performs better for programs like `m88ksim`, `espresso`, and `bison` because the size of the popular call graph for these applications is larger than the size of the instruction cache. This allows our algorithm to fully exploit cache line coloring, arriving at a layout that significantly reduces the number of first-generation cache conflicts.

For programs such as `flex` and `gzip`, the reason why our algorithm and the P&H algorithm have approximately the same miss rate can be seen by looking at the partitioning part of our algorithm. Here, the program is partitioned into popular and unpopular procedures and edges. In performing this partitioning, these programs are split into disjoint subgraphs where most of the subgraphs are smaller than the size of the cache. Since these popular subgraphs easily fit within the instruction cache, we can arbitrarily map their procedures. For example, `gzip` visits only a small number of very popular procedures when processing the input file `gcc-2.7.2.tar`. This is seen in Table 1, where the size of the popular procedures for `gzip` amount to only 10K (3% of the total executable size), and the simulated instruction cache size we used is 8K. For applications where the popular subgraphs fit within the size of the instruction cache, our color mapping algorithm and the Pettis and Hansen algorithm will have similar performance.

Table 2 shows that for `eqntott`, the instruction cache miss rate when applying the P&H mapping is larger than the miss rate of the original mapping. This effect occurs for two reasons. One reason is the poor choice made by the P&H algorithm when merging chains that sum to a size larger than the instruction cache, creating cache conflicts within the newly merged chain. The second reason is that both our algorithm and the P&H algorithm only model first-generation conflicts in the call graphs. The call graph used in this study only models the frequency of procedure calls between a procedure and its direct children. It does not model the temporal locality between a procedure and all of the procedures that it can possibly reach in the call graph, and any of these reachable procedures can cause cache conflicts. This emphasizes the fact that finding an optimal mapping to minimize conflicts is NP-complete [10]. In the next section we suggest further optimizations to our algorithm in order to address misses beyond first-generation cache conflicts.

The results in Table 2 are all gathered using the same input that was also used to profile the program. An important issue involving profiled-based optimizations is how well does a single input capture the typical behavior of future runs of the program. Several researchers have investigated this problem and have found that programs have predictable behavior between different inputs [4, 7, 19]. Even so, care must be taken when choosing the inputs to guide optimizations. In this vein, we took a few of the optimized programs used to produce the results in Table 2 and ran them using different inputs. Table 3 shows the cache miss rates for these programs using different inputs. For these different inputs, the results show that a similar reduction in miss rate of 45% is achieved when comparing our color mapping algorithm to the original layout, and the reduction in miss rate for our algorithm when compared to P&H is about the same at 15%. In general, when examining different inputs our algorithm still shows significant reductions in the original instruction cache miss ratios, while consistently showing an advantage over P&H.

Program	Input	I-Cache Miss Rate			# Instruction Cache Misses		
		Original	P&H	Color	Original	P&H	Color
li	li_short	1.2%	0.3%	0.3%	19,590,740	4,947,711	4,165,895
m88ksim	dhry.lit	4.3%	2.9%	2.3%	2,165,135,958	1,435,239,250	1,144,552,965
espresso	Z5xp1	1.3%	1.3%	0.9%	383,390	377,091	256,792
	bca	0.2%	0.2%	0.1%	1,306,820	1,255,329	681,001
	cps	0.4%	0.4%	0.3%	2,514,798	2,471,459	1,659,891
	dc1	2.6%	2.5%	2.1%	23,570	22,917	19,306
	mlp4	1.1%	1.1%	0.8%	944,483	930,846	659,793
	opa	0.6%	0.6%	0.4%	883,401	867,858	644,822
	ti	0.5%	0.5%	0.3%	3,874,952	3,769,353	2,691,281
bison	c-parse.y	1.7%	1.7%	1.3%	816,872	800,556	604,087
flex	unfixit.l	2.7%	2.1%	2.1%	327,781	254,345	250,005
gzip	bison-1.25.tar	1.1%	0%	0%	4,152,085	33,950	1,961
Average		2.0%	1.3%	1.1%			

Table 3: Instruction cache performance using multiple inputs for the Original mapping, Pettis and Hansen (P&H) mapping, and our Color mapping algorithm. In calculating the overall average, a value for `espresso` is included only once, which is the average miss rate for `espresso` on all of the inputs shown.

To examine the impact procedure reordering optimizations have on the performance of these programs, Figures 5 and 6 show the estimated performance in instructions issued per cycle (IPC) for the original program, P&H mapping, and our color algorithm for two different architectures. The higher the IPC the better. For these results we assume each instruction takes one cycle to execute, and that the only pipeline stalls are due to misses in the instruction cache. Figure 5 shows a conservative estimate of performance using a single issue architecture with a small (5 cycle) first-level instruction cache miss penalty. Figure 6 shows an aggressive 4-way issue architecture with a larger (10 cycle) first-level instruction cache miss penalty. The results in Figure 5 show that for a conservative architecture our color mapping algorithm increases the IPC on average by 5% when compared to the original mapping, and by 1% when compared to P&H. The results in Figure 6 show that for a more aggressive architecture that our color mapping algorithm increases the IPC on average by 26% when compared to the original mapping, and by 6% when compared to P&H. These two graphs show the potential increase in performance when using our algorithm. As seen in the two figures, the performance for our algorithm is the same for `li`, `flex`, and `gzip` when compared to P&H for the reasons discussed in the previous paragraphs. For programs like `m88ksim`, `espresso`, and `bison` which have larger more complicated call graphs, the figures show that the increase in performance for our algorithm is 2% to 13% when compared to P&H.

One issue to consider with our algorithm is that in order to avoid first-generation cache conflicts our color mapping will insert space into compound nodes as described in §2.1. This space is later filled with unpopular procedures. This could possibly have two adverse effects. The first is, if no unpopular procedure

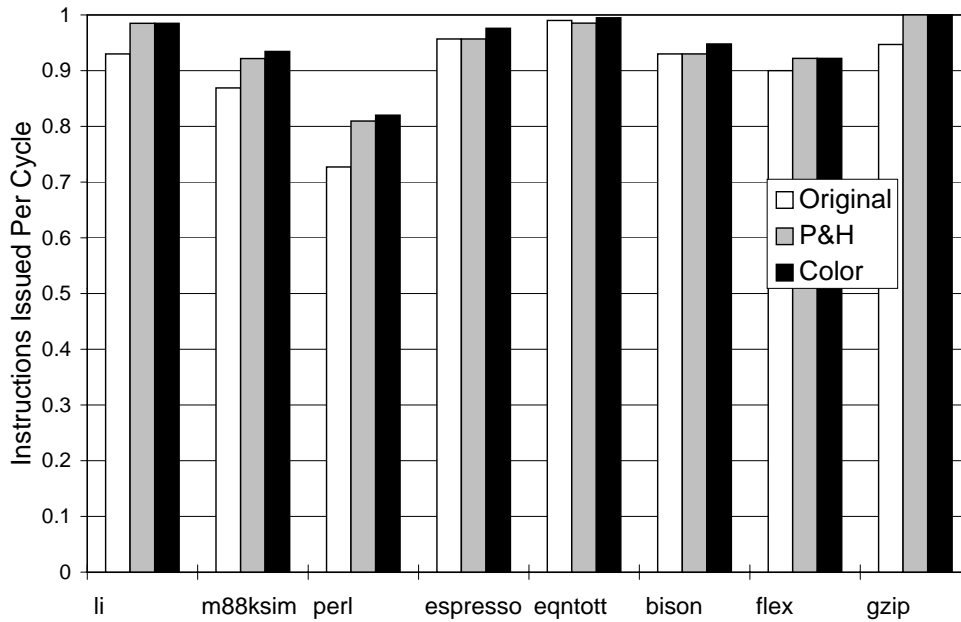


Figure 5: Instructions issued per cycle for a single issue architecture, with an 8K direct mapped instruction cache which has a 5 cycle cache miss penalty.

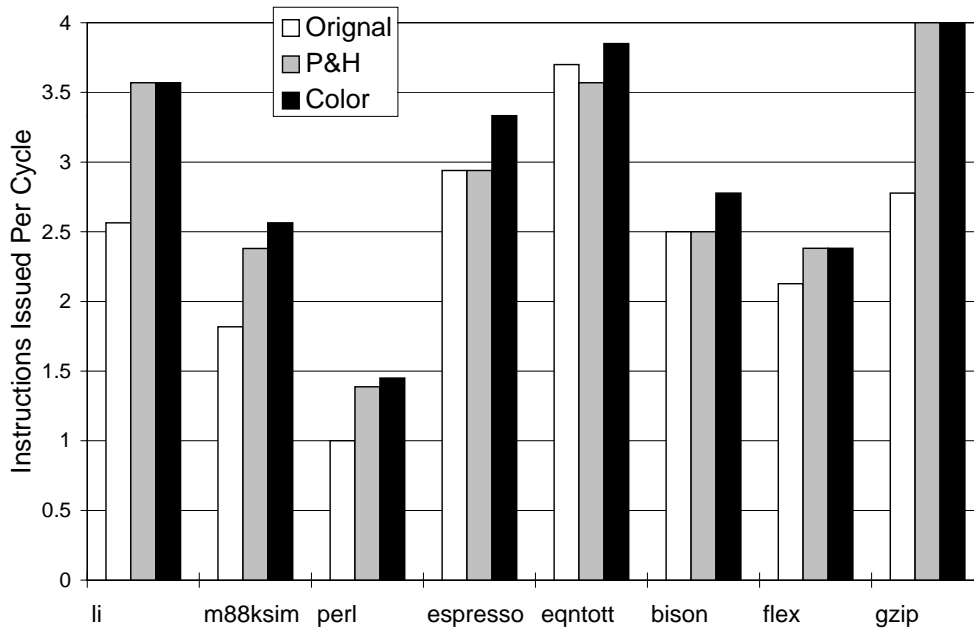


Figure 6: Instructions issued per cycle for a 4-way issue processor with an 8K direct mapped instruction cache which has a 10 cycle cache miss penalty.

can be found when trying to fill a space, then this could result in an increase in the executable size. For the programs we examined this was never an issue. As seen in Table 1, on average only 8% of the procedures were labeled as popular, leaving more than enough unpopular procedures to fill in any gaps that were created by the color mapping algorithm. The second effect is that the size of the working set of the program may increase due to the algorithm filling spaces in the compound nodes with unpopular procedures. From our results we do not believe this will be an issue, but further investigation is needed. When performing the color mapping for the programs we examined, on average only 3K worth of unpopular procedures were used as filler and inserted into the popular color mapping, as seen in Table 1. Since the average size for all of the popular procedures in a program was 33K, this increases the size of the popular mapping section of the address space by only 8%.

5 Discussion and Future Work

In this section we discuss how to apply our color mapping algorithm to associative caches, describe how our algorithm can benefit from basic block reordering and procedure splitting, and describe future work on how to improve the performance of our algorithm by using more information on temporal locality to guide the mapping.

5.1 Color Mapping for Associative Caches

In this paper only described our algorithm as applied to direct mapped caches and examined its performance for an 8K direct mapped instruction cache. Our algorithm can easily be applied to set-associative instruction caches. To accomplish this, we treat the associativity of the cache as another dimension in the mapping of the address space. For associative caches our algorithm breaks up the address space into chunks, equal in size to (*the number of cache sets * the cache line size*). Therefore, the number of sets represents the number of available colors in the mapping. The color mapping algorithm can then be applied as described in §2.1, with only a few minor changes. The algorithm changes slightly to keep track of the number of times each color (set) appears in the procedure's unavailable-set of colors. Therefore, mapping a procedure to a color (set) does not cause any conflicts as long as the number of times that color (set) appears in the unavailable-set of colors is less than the degree of associativity of the cache. This effectively turns the unavailable-set into a *multiset*, which allows each color to appear in the set up to the associativity of the cache.

5.2 Color Mapping with Basic Block Reordering and Procedure Splitting

The results in §4 do not show the full potential of our coloring algorithm, since our algorithm can benefit from other code reordering techniques such as basic block reordering and procedure splitting [8, 11]. Our color mapping algorithm can benefit from basic block reordering because once the basic blocks have been aligned and condensed into the first part of the procedure, the cache line colors used by the frequently executed basic blocks are the only colors we have to worry about when performing the procedure mapping. Using basic block profiling, each procedure would contain two sets of cache colors: those for the important portions of the procedure, and those for the unimportant. Then the only basic blocks we need to worry about in the unavailable-set of colors are the important basic blocks.

Performing procedure splitting can also be used to improve the performance of our color mapping algorithm. This can be achieved by performing procedure splitting to help reduce the coloring constraints between different procedures. For example, if half of a procedure X , $X1$, calls a procedure Y , and the other half of the procedure X , $X2$, calls procedure Z , then finding a location for X in the color mapping as described in §2.1 will have to try and avoid the colors used by both Y and Z . If procedure splitting is performed so that X is split into two separate procedures $X1$ and $X2$, then this can help reduce the coloring constraints on X . After procedure X is split into $X1$ and $X2$, the color mapping for $X1$ only needs to avoid colors used by $X2$ and Y , and the color mapping for $X2$ needs to only avoid colors used by $X1$ and Z . This can help free up coloring constraints for very large procedures and procedures that have a significant number of different call destinations.

5.3 Using Improved Temporal Locality Data

Our color mapping algorithm, as described in §2.1, concentrates on eliminating conflicts between edges in the control flow graph. For our results, these edges happen to be first-generation cache conflicts because the graph edges represent the call edges between a procedure and its direct parents and children. Our algorithm can easily be applied to more detailed forms of profile and trace information by adding extra edges between procedures, treating these edges as a second set of procedure call edges in our color mapping algorithm. These additional edges, with the appropriate weights, can then be used in the unavailable-set of colors in order to further eliminate cache conflicts.

The call graph and profiles we used to guide the mappings do not provide enough information to determine the temporal locality for a depth greater than one procedure call (first-generation) in the graph. Even for first-generation misses, a call graph does not provide exact information about temporal locality. Therefore, our algorithm tries to remove the *worse case* number of first-generation misses. For example, in Figure 1, we know that since the edge $C \rightarrow D$ was executed 70 times, that if C and D had overlapping cache lines, then the call to D and the return to C could in the worst case cause $((70+70)*number\ of\ overlapping\ cache\ lines)$

misses. For future work we are using control flow analysis of the program’s structure to indicate if all the calls from $C \rightarrow D$ were done during one invocation of C or whether they were spread out over several invocations, similar to the control flow analysis used by McFarling [10]. We are also using control flow analysis to determine how much of procedure C can actually overlap with procedure D , so we only have to include those cache lines in D ’s unavailable-set of colors. This will help provide more accurate temporal locality information for first-generation conflicts, but it does not provide the additional temporal locality information we would like for deeper paths in the call graph.

When profiling just the call edges, there is no way to get a good indication of temporal locality for a path longer than one procedure call edge. For example, in Figure 1 we have no way of knowing for the call edge $C \rightarrow D$ how many of the procedure calls to D came down the path through procedure B and how many went through procedure E , nor do we know how much temporal locality there is between B and D or E and D . Some of this information can be obtained by using full path profiling, which would allow one to know the frequency of each path [1, 20], although full path profiling still does not provide optimal temporal locality information. One way to obtain additional information on temporal locality is to store the full trace of a program. Capturing, storing, and processing a full trace can be very time and space consuming, but efficient techniques have been proposed to capture and process this information in a compact form, such as the gap model proposed by Quong [14]. We plan on investigating the use of full path profiling and the gap model with our color mapping algorithm in order to eliminate additional cache conflicts for deeper paths in the call graph.

6 Conclusions

The performance of the cache-based memory system is critical in today’s processors. Research has shown that compiler optimizations can significantly reduce this latency, and every opportunity should be taken by the compiler to do so.

The contribution of this paper is a new algorithm for procedure mapping which takes into consideration the call graph, procedure size, cache size, and cache line size. An improved algorithm is achieved by keeping track of the cache blocks (colors) used by each procedure as it is mapped, in order to avoid cache conflicts. This color mapping allows our algorithm to intelligently place unmapped procedures, and to efficiently move a procedure that has already been mapped, by preserving prior color dependencies with that procedure’s parents and children in the call graph. This provides our main advantage over prior work, in that we can accurately map procedures in a popular call graph even if the size of the graph is larger than the size of the instruction cache. This ability is very important, especially for applications which have large and complicated control flow graphs, which result in large instruction cache miss rates due to conflict misses. Our results showed that we were able to reduce the cache miss rate on average by 45% over the original

procedure mapping. In comparison to prior work, our algorithm reduced the cache miss rate on average 14% below that of the Pettis and Hansen algorithm [11].

In this study we concentrated on applying our color mapping algorithm to procedure reordering. Our algorithm can be combined and benefit from other code reordering techniques such as basic block reordering, taking into consideration looping structures, and procedure splitting. These are topics of future research. In this paper we also concentrated on the performance achieved using call edge profiles to guide the optimizations in order to eliminate first-generation cache conflicts. We are currently investigating how to apply our algorithm to use full path profiling and other trace collection techniques in order to collect improved temporal locality information. We are also examining how to apply our color mapping algorithm to statically formed call graphs using static program estimation.

Acknowledgments

We would like to thank Amitabh Srivastava and Alan Eustace for providing ATOM, which greatly simplified our work, and Jeffrey Dean, Alan Eustace, Waleed Meleis, and Russell Quong for providing useful suggestions and comments on this paper. Brad Calder was supported by Digital Equipment Corporation's Western Research Lab. David Kaeli was supported by an NSF CAREER Program award No. 9501172.

References

- [1] T. Ball and J. Larus. Efficient path profiling. In *29th International Symposium on Microarchitecture*, December 1996.
- [2] L. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
- [3] B.N. Bershad, D. Lee, T.H. Romer, and J.B. Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *Six International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 158–170, October 1994.
- [4] B. Calder, D. Grunwald, and A. Srivastava. The predictability of branches in libraries. In *28th International Symposium on Microarchitecture*, pages 24–34, Ann Arbor, MI, November 1995. IEEE.
- [5] B. Calder, D. Grunwald, and B. Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4), 1994.
- [6] P.J. Denning and S. C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3):191–198, March 1972.

- [7] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 85–95, Boston, Mass., October 1992. ACM.
- [8] 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.
- [9] D. Kaeli. Issues in trace-driven simulation. Lecture Notes in Computer Science No. 729, Performance Evaluation of Computer and Communication Systems, L. Donatiello and R. Nelson eds., Springer-Verlag, 1993, pp. 224-244., 1990.
- [10] 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.
- [11] K. Pettis and R.C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, ACM, June 1990.
- [12] S.A. Przybylski. *Cache Design: A Performance-Directed Approach*. Morgan Kaufmann, San Mateo, CA, 1990.
- [13] T.R. Puzak. Analysis of cache replacement-algorithms. Ph.D. Dissertation, University of Massachusetts, Amherst MA, 1985.
- [14] R.W. Quong. Expected I-cache miss rates via the gap model. In *21st Annual International Symposium on Computer Architecture*, pages 372–383, April 1994.
- [15] A.D. Samples and P.N. Hilfinger. Code reorganization for instruction caches. Technical Report UCB/CSD 88/447, October 1988.
- [16] 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.
- [17] J.G. Thompson. Efficient analysis of caching systems. Ph.D. Dissertation, University of California, Berkeley, 1987.

- [18] J. Torrellas, C. Xia, and R. Daigle. Optimizing instruction cache performance for operating system intensive workloads. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 360–369, January 1995.
- [19] D.W. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 59–70, Toronto, Ontario, Canada, June 1991.
- [20] C. Young, N. Gloy, and M.D. Smith. A comparative analysis of schemes for correlated branch prediction. In *22nd Annual International Symposium on Computer Architecture*, pages 276–286, June 1995.

WRL Research Reports

- “Titan System Manual.” **Michael J. K. Nielsen.** WRL Research Report 86/1, September 1986.
- “Global Register Allocation at Link Time.” **David W. Wall.** WRL Research Report 86/3, October 1986.
- “Optimal Finned Heat Sinks.” **William R. Hamburgen.** WRL Research Report 86/4, October 1986.
- “The Mahler Experience: Using an Intermediate Language as the Machine Description.” **David W. Wall and Michael L. Powell.** WRL Research Report 87/1, August 1987.
- “The Packet Filter: An Efficient Mechanism for User-level Network Code.” **Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.** WRL Research Report 87/2, November 1987.
- “Fragmentation Considered Harmful.” **Christopher A. Kent, Jeffrey C. Mogul.** WRL Research Report 87/3, December 1987.
- “Cache Coherence in Distributed Systems.” **Christopher A. Kent.** WRL Research Report 87/4, December 1987.
- “Register Windows vs. Register Allocation.” **David W. Wall.** WRL Research Report 87/5, December 1987.
- “Editing Graphical Objects Using Procedural Representations.” **Paul J. Asente.** WRL Research Report 87/6, November 1987.
- “The USENET Cookbook: an Experiment in Electronic Publication.” **Brian K. Reid.** WRL Research Report 87/7, December 1987.
- “MultiTitan: Four Architecture Papers.” **Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.** WRL Research Report 87/8, April 1988.
- “Fast Printed Circuit Board Routing.” **Jeremy Dion.** WRL Research Report 88/1, March 1988.
- “Compacting Garbage Collection with Ambiguous Roots.” **Joel F. Bartlett.** WRL Research Report 88/2, February 1988.
- “The Experimental Literature of The Internet: An Annotated Bibliography.” **Jeffrey C. Mogul.** WRL Research Report 88/3, August 1988.
- “Measured Capacity of an Ethernet: Myths and Reality.” **David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.** WRL Research Report 88/4, September 1988.
- “Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.” **Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.** WRL Research Report 88/5, December 1988.
- “SCHEME->C A Portable Scheme-to-C Compiler.” **Joel F. Bartlett.** WRL Research Report 89/1, January 1989.
- “Optimal Group Distribution in Carry-Skip Adders.” **Silvio Turrini.** WRL Research Report 89/2, February 1989.
- “Precise Robotic Paste Dot Dispensing.” **William R. Hamburgen.** WRL Research Report 89/3, February 1989.
- “Simple and Flexible Datagram Access Controls for Unix-based Gateways.” **Jeffrey C. Mogul.** WRL Research Report 89/4, March 1989.
- “Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.” **V. Srinivasan and Jeffrey C. Mogul.** WRL Research Report 89/5, May 1989.
- “Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.” **Norman P. Jouppi and David W. Wall.** WRL Research Report 89/7, July 1989.
- “A Unified Vector/Scalar Floating-Point Architecture.” **Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.** WRL Research Report 89/8, July 1989.

- “Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.” **Norman P. Jouppi.** WRL Research Report 89/9, July 1989.
- “Integration and Packaging Plateaus of Processor Performance.” **Norman P. Jouppi.** WRL Research Report 89/10, July 1989.
- “A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.” **Norman P. Jouppi and Jeffrey Y. F. Tang.** WRL Research Report 89/11, July 1989.
- “The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.” **Norman P. Jouppi.** WRL Research Report 89/13, July 1989.
- “Long Address Traces from RISC Machines: Generation and Analysis.” **Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.** WRL Research Report 89/14, September 1989.
- “Link-Time Code Modification.” **David W. Wall.** WRL Research Report 89/17, September 1989.
- “Noise Issues in the ECL Circuit Family.” **Jeffrey Y.F. Tang and J. Leon Yang.** WRL Research Report 90/1, January 1990.
- “Efficient Generation of Test Patterns Using Boolean Satisfiability.” **Tracy Larrabee.** WRL Research Report 90/2, February 1990.
- “Two Papers on Test Pattern Generation.” **Tracy Larrabee.** WRL Research Report 90/3, March 1990.
- “Virtual Memory vs. The File System.” **Michael N. Nelson.** WRL Research Report 90/4, March 1990.
- “Efficient Use of Workstations for Passive Monitoring of Local Area Networks.” **Jeffrey C. Mogul.** WRL Research Report 90/5, July 1990.
- “A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.” **John S. Fitch.** WRL Research Report 90/6, July 1990.
- “1990 DECWRL/Livermore Magic Release.” **Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.** WRL Research Report 90/7, September 1990.
- “Pool Boiling Enhancement Techniques for Water at Low Pressure.” **Wade R. McGillis, John S. Fitch, William R. Hamburgren, Van P. Carey.** WRL Research Report 90/9, December 1990.
- “Writing Fast X Servers for Dumb Color Frame Buffers.” **Joel McCormack.** WRL Research Report 91/1, February 1991.
- “A Simulation Based Study of TLB Performance.” **J. Bradley Chen, Anita Borg, Norman P. Jouppi.** WRL Research Report 91/2, November 1991.
- “Analysis of Power Supply Networks in VLSI Circuits.” **Don Stark.** WRL Research Report 91/3, April 1991.
- “TurboChannel T1 Adapter.” **David Boggs.** WRL Research Report 91/4, April 1991.
- “Procedure Merging with Instruction Caches.” **Scott McFarling.** WRL Research Report 91/5, March 1991.
- “Don’t Fidget with Widgets, Draw!” **Joel Bartlett.** WRL Research Report 91/6, May 1991.
- “Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.” **Wade R. McGillis, John S. Fitch, William R. Hamburgren, Van P. Carey.** WRL Research Report 91/7, June 1991.
- “Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.” **G. May Yip.** WRL Research Report 91/8, June 1991.
- “Interleaved Fin Thermal Connectors for Multichip Modules.” **William R. Hamburgren.** WRL Research Report 91/9, August 1991.
- “Experience with a Software-defined Machine Architecture.” **David W. Wall.** WRL Research Report 91/10, August 1991.

- “Network Locality at the Scale of Processes.” **Jeffrey C. Mogul**. WRL Research Report 91/11, November 1991.
- “Cache Write Policies and Performance.” **Norman P. Jouppi**. WRL Research Report 91/12, December 1991.
- “Packaging a 150 W Bipolar ECL Microprocessor.” **William R. Hamburgren, John S. Fitch**. WRL Research Report 92/1, March 1992.
- “Observing TCP Dynamics in Real Networks.” **Jeffrey C. Mogul**. WRL Research Report 92/2, April 1992.
- “Systems for Late Code Modification.” **David W. Wall**. WRL Research Report 92/3, May 1992.
- “Piecewise Linear Models for Switch-Level Simulation.” **Russell Kao**. WRL Research Report 92/5, September 1992.
- “A Practical System for Intermodule Code Optimization at Link-Time.” **Amitabh Srivastava and David W. Wall**. WRL Research Report 92/6, December 1992.
- “A Smart Frame Buffer.” **Joel McCormack & Bob McNamara**. WRL Research Report 93/1, January 1993.
- “Recovery in Spritely NFS.” **Jeffrey C. Mogul**. WRL Research Report 93/2, June 1993.
- “Tradeoffs in Two-Level On-Chip Caching.” **Norman P. Jouppi & Steven J.E. Wilton**. WRL Research Report 93/3, October 1993.
- “Unreachable Procedures in Object-oriented Programming.” **Amitabh Srivastava**. WRL Research Report 93/4, August 1993.
- “An Enhanced Access and Cycle Time Model for On-Chip Caches.” **Steven J.E. Wilton and Norman P. Jouppi**. WRL Research Report 93/5, July 1994.
- “Limits of Instruction-Level Parallelism.” **David W. Wall**. WRL Research Report 93/6, November 1993.
- “Fluoroelastomer Pressure Pad Design for Microelectronic Applications.” **Alberto Makino, William R. Hamburgren, John S. Fitch**. WRL Research Report 93/7, November 1993.
- “A 300MHz 115W 32b Bipolar ECL Microprocessor.” **Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgren, Russell Kao, and Richard Swan**. WRL Research Report 93/8, December 1993.
- “Link-Time Optimization of Address Calculation on a 64-bit Architecture.” **Amitabh Srivastava, David W. Wall**. WRL Research Report 94/1, February 1994.
- “ATOM: A System for Building Customized Program Analysis Tools.” **Amitabh Srivastava, Alan Eustace**. WRL Research Report 94/2, March 1994.
- “Complexity/Performance Tradeoffs with Non-Blocking Loads.” **Keith I. Farkas, Norman P. Jouppi**. WRL Research Report 94/3, March 1994.
- “A Better Update Policy.” **Jeffrey C. Mogul**. WRL Research Report 94/4, April 1994.
- “Boolean Matching for Full-Custom ECL Gates.” **Robert N. Mayo, Herve Touati**. WRL Research Report 94/5, April 1994.
- “Software Methods for System Address Tracing: Implementation and Validation.” **J. Bradley Chen, David W. Wall, and Anita Borg**. WRL Research Report 94/6, September 1994.
- “Performance Implications of Multiple Pointer Sizes.” **Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava**. WRL Research Report 94/7, December 1994.
- “How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.” **Keith I. Farkas, Norman P. Jouppi, and Paul Chow**. WRL Research Report 94/8, December 1994.

- “Drip: A Schematic Drawing Interpreter.” **Ramsey W. Haddad**. WRL Research Report 95/1, March 1995.
- “Recursive Layout Generation.” **Louis M. Monier, Jeremy Dion**. WRL Research Report 95/2, March 1995.
- “Contour: A Tile-based Gridless Router.” **Jeremy Dion, Louis M. Monier**. WRL Research Report 95/3, March 1995.
- “The Case for Persistent-Connection HTTP.” **Jeffrey C. Mogul**. WRL Research Report 95/4, May 1995.
- “Network Behavior of a Busy Web Server and its Clients.” **Jeffrey C. Mogul**. WRL Research Report 95/5, October 1995.
- “The Predictability of Branches in Libraries.” **Brad Calder, Dirk Grunwald, and Amitabh Srivastava**. WRL Research Report 95/6, October 1995.
- “Shared Memory Consistency Models: A Tutorial.” **Sarita V. Adve, Kourosh Gharachorloo**. WRL Research Report 95/7, September 1995.
- “Eliminating Receive Livelock in an Interrupt-driven Kernel.” **Jeffrey C. Mogul and K. K. Ramakrishnan**. WRL Research Report 95/8, December 1995.
- “Memory Consistency Models for Shared-Memory Multiprocessors.” **Kourosh Gharachorloo**. WRL Research Report 95/9, December 1995.
- “Register File Design Considerations in Dynamically Scheduled Processors.” **Keith I. Farkas, Norman P. Jouppi, Paul Chow**. WRL Research Report 95/10, November 1995.
- “Optimization in Permutation Spaces.” **Silvio Turrini**. WRL Research Report 96/1, November 1996.
- “Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory.” **Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath**. WRL Research Report 96/2, November 1996.
- “Efficient Procedure Mapping using Cache Line Coloring.” **Amir H. Hashemi, David R. Kaeli, and Brad Calder**. WRL Research Report 96/3, October 1996.
- “Optimizations and Placement with the Genetic Workbench.” **Silvio Turrini**. WRL Research Report 96/4, November 1996.

WRL Technical Notes

- “TCP/IP PrintServer: Print Server Protocol.” **Brian K. Reid and Christopher A. Kent.** WRL Technical Note TN-4, September 1988.
- “TCP/IP PrintServer: Server Architecture and Implementation.” **Christopher A. Kent.** WRL Technical Note TN-7, November 1988.
- “Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.” **Joel McCormack.** WRL Technical Note TN-9, September 1989.
- “Why Aren’t Operating Systems Getting Faster As Fast As Hardware?.” **John Ousterhout.** WRL Technical Note TN-11, October 1989.
- “Mostly-Copying Garbage Collection Picks Up Generations and C++.” **Joel F. Bartlett.** WRL Technical Note TN-12, October 1989.
- “Characterization of Organic Illumination Systems.” **Bill Hamburg, Jeff Mogul, Brian Reid, Alan Eustace, Richard Swan, Mary Jo Doherty, and Joel Bartlett.** WRL Technical Note TN-13, April 1989.
- “Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.” **Norman P. Jouppi.** WRL Technical Note TN-14, March 1990.
- “Limits of Instruction-Level Parallelism.” **David W. Wall.** WRL Technical Note TN-15, December 1990.
- “The Effect of Context Switches on Cache Performance.” **Jeffrey C. Mogul and Anita Borg.** WRL Technical Note TN-16, December 1990.
- “MTOOL: A Method For Detecting Memory Bottlenecks.” **Aaron Goldberg and John Hennessy.** WRL Technical Note TN-17, December 1990.
- “Predicting Program Behavior Using Real or Estimated Profiles.” **David W. Wall.** WRL Technical Note TN-18, December 1990.
- “Cache Replacement with Dynamic Exclusion.” **Scott McFarling.** WRL Technical Note TN-22, November 1991.
- “Boiling Binary Mixtures at Subatmospheric Pressures.” **Wade R. McGillis, John S. Fitch, William R. Hamburg, Van P. Carey.** WRL Technical Note TN-23, January 1992.
- “A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach.” **John S. Fitch.** WRL Technical Note TN-24, January 1992.
- “TurboChannel Versatec Adapter.” **David Boggs.** WRL Technical Note TN-26, January 1992.
- “A Recovery Protocol For Spritely NFS.” **Jeffrey C. Mogul.** WRL Technical Note TN-27, April 1992.
- “Electrical Evaluation Of The BIPS-0 Package.” **Patrick D. Boyle.** WRL Technical Note TN-29, July 1992.
- “Transparent Controls for Interactive Graphics.” **Joel F. Bartlett.** WRL Technical Note TN-30, July 1992.
- “Design Tools for BIPS-0.” **Jeremy Dion & Louis Monier.** WRL Technical Note TN-32, December 1992.
- “Link-Time Optimization of Address Calculation on a 64-Bit Architecture.” **Amitabh Srivastava and David W. Wall.** WRL Technical Note TN-35, June 1993.
- “Combining Branch Predictors.” **Scott McFarling.** WRL Technical Note TN-36, June 1993.
- “Boolean Matching for Full-Custom ECL Gates.” **Robert N. Mayo and Herve Touati.** WRL Technical Note TN-37, June 1993.
- “Piecewise Linear Models for Rsim.” **Russell Kao, Mark Horowitz.** WRL Technical Note TN-40, December 1993.

“Speculative Execution and Instruction-Level Parallelism.” **David W. Wall.** WRL Technical Note TN-42, March 1994.

“Ramonamap - An Example of Graphical Groupware.” **Joel F. Bartlett.** WRL Technical Note TN-43, December 1994.

“ATOM: A Flexible Interface for Building High Performance Program Analysis Tools.” **Alan Eustace and Amitabh Srivastava.** WRL Technical Note TN-44, July 1994.

“Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS.” **Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.** WRL Technical Note TN-45, March 1994.

“Experience with a Wireless World Wide Web Client.” **Joel F. Bartlett.** WRL Technical Note TN-46, March 1995.

“I/O Component Characterization for I/O Cache Designs.” **Kathy J. Richardson.** WRL Technical Note TN-47, April 1995.

“Attribute caches.” **Kathy J. Richardson, Michael J. Flynn.** WRL Technical Note TN-48, April 1995.

“Operating Systems Support for Busy Internet Servers.” **Jeffrey C. Mogul.** WRL Technical Note TN-49, May 1995.

“The Predictability of Libraries.” **Brad Calder, Dirk Grunwald, Amitabh Srivastava.** WRL Technical Note TN-50, July 1995.

WRL Research Reports and Technical Notes are available on the World Wide Web, from <http://www.research.digital.com/wrl/techreports/index.html>.