Reducing Branch Costs via Branch Alignment

Brad Calder and Dirk Grunwald *

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309–0430 USA
{calder,grunwald}@cs.colorado.edu

Abstract

Several researchers have proposed algorithms for basic block reordering. We call these *branch alignment* algorithms. The primary emphasis of these algorithms has been on improving instruction cache locality, and the few studies concerned with branch prediction reported small or minimal improvements. As wide-issue architectures become increasingly popular the importance of reducing branch costs will increase, and branch alignment is one mechanism which can effectively reduce these costs.

In this paper, we propose an improved branch alignment algorithm that takes into consideration the architectural cost model and the branch prediction architecture when performing the basic block reordering. We show that branch alignment algorithms can improve a broad range of static and dynamic branch prediction architectures. We also show that a programs performance can be improved by approximately 5% even when using recently proposed, highly accurate branch prediction architectures. The programs are compiled by any existing compiler and then transformed via binary transformations. When implementing these algorithms on a Alpha AXP 21604 up to a 16% reduction in total execution time is achieved.

Keywords: Branch Prediction, Profile-based Optimization, Branch Target Buffers, Trace Scheduling.

1 Introduction

Conventional processor architectures, particularly modern superscalar designs, are extremely sensitive to control flow changes. Changes in control flow, be they conditional or unconditional branches, direct or indirect function calls or returns, are not detected until those instructions are decoded. To keep the pipeline fully utilized, processors typically fetch the address following the most recent address. If the decoded instruction breaks the control flow, the previously fetched instruction can not be used, and a new instruction must be fetched, introducing a "pipeline bubble" or unused pipeline step. This is called an instruction *misfetch penalty*.

The final destination for conditional branches, indirect function calls and returns are typically not available until a latter stage of the pipeline. At this point the branch has been completely evaluated. The processor may elect to fetch and decode instructions on the assumption that the eventual branch target can be accurately predicted. If the processor mispredicts the branch destination, the instructions fetched from the incorrect instruction stream must be discarded, leading to several "pipeline bubbles" and causing a *mispredict penalty*. In practice, pipeline bubbles due to mispredicted breaks in control flow degrade a programs performance more than the misfetch penalty. For example, the combined branch mispredict penalty for the Digital Alpha AXP 21064 processor is ten instructions. By comparison, the AXP 21064 would lose only two instructions from instruction misfetches.

Almost all modern architectures use some form of branch prediction, and reducing the number of misfetch and misprediction penalties will be increasingly important for wide-issue architectures. In this paper, we examine algorithms that reorder the structure of a program to improve the accuracy of the branch fetch and prediction architectures. Our code transformations reduce the number of mispredicted branches and the number of misfetched instructions. Essentially, the method is this: we restructure the control flow graph so that *fall-through* branches occur more frequently. We use profile information to direct the transformation, and an architectural cost model to decide if the transformation is warranted. Transformations include rearranging the placement of basic blocks, changing the sense of conditional operations, moving unconditional branches out of the frequently executed path, and occasionally inserting unconditional branches.

We show that static and dynamic branch prediction mechanisms we examine benefit from such transformations, with the static branch architectures benefiting more than the dynamic. We implemented the branch alignment binary transformation algorithms on a DEC Alpha AXP 21064, and measured total execution time improvements of up to 16%.

*To appear in the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI), San Jose, California. October 1994.

2 Prior Work

There has been considerable work on profile-driven program optimization. In this section we consider relevant work on optimizations for instruction caches and branch mechanisms.

Optimization for Memory Hierarchies: Due to the expense of memory on early computers, much early work focused on reducing paging in virtual memory systems. Several researchers explored ways to group related subroutines or basic blocks onto the same virtual memory pages [1, 8, 11, 13, 10]. Other researchers extended this work to lower levels of the memory hierarchy, optimizing the performance of instruction caches. McFarling [15] described an algorithm to reduce instruction cache conflicts for a particular class of programs. Hwu and Chang [18] describe a more general and more effective technique using compile time analysis in the IMPACT-I compiler system. Using profile-based transformations, the IMPACT-I compiler inlines subroutines and performs trace analysis. For each subroutine, instructions are packed using the most frequently executed traces, moving infrequently executed traces to the end of the function. Following this, global analysis arranges functions to reduce inter-function cache conflicts. Similar transformations were applied by Pettis and Hansen [21] for programs on the HP PA-RISC.

Optimizations for Control Flow: McFarling and Hennessy [17] described a number of methods to reduce branch misprediction and instruction fetch penalties, including profile-driven static branch prediction, delay slots and a form of branch alignment. Their variant of branch alignment only considered `if/then/else` constructs. Later, Bray and Flynn [4] extended the work of McFarling *et al* while examining various branch target buffer (BTB) architectures. Yet, they also only examined `if/then/else` constructs.

Yeh *et al* [26] commented that with trace scheduling, taken branches could only be reduced from $\approx$ 62% of the executed conditional branches to $\approx$ 50% of executed conditional branches. The earlier study by Hwu and Chang [18] showed a $\approx$ 58% fall-through rate after branch alignment. The papers by McFarling and Hennessy, Bray and Flynn, and Pettis and Hansen did not report the change in the percentage of taken branches.

The branch alignment reordering algorithm proposed by Hwu *et al* is more general than McFarling's and Bray's. Hwu and Chang examined all basic blocks, rearranging them to achieve a better branch alignment. They were able to handle branches that do not form an `if/then/else` structure. The work by Pettis and Hansen [21] describes a greedy algorithm for branch alignment which is similar to Hwu and Chang's since they look at all basic blocks. The Pettis and Hansen greedy algorithm is more general than the Hwu and Chang algorithm, and performs better in terms of reducing the cost of branches.

In this paper we describe an algorithm which is an extension of the Pettis and Hansen algorithm, and we compare our results to their greedy algorithm. We also improve upon the analysis and the effectiveness of branch alignment over that of McFarling, Bray and Flynn, Hwu and Chang, and Pettis and Hansen. We describe how to efficiently apply branch alignment to various static and dynamic prediction architectures and we measure the effectiveness of branch alignment on these architectures.

Our technique is similar to the methods of McFarling, Bray and Flynn, Hwu and Chang, and Pettis and Hansen. However, we do not inline functions, perform global analysis or duplicate code. We perform the analysis using an object code post-processor rather than a compiler. This simplifies the analysis and avoids recompiling the full program for these simple transformations. This also allows us to apply branch alignment to the full program, including portions normally not compiled by the user, such as program libraries, and to process many programs generated by a number of different compilers. With such a post-processor tool, branch alignment would normally be only one of several optimizations applied to the program.

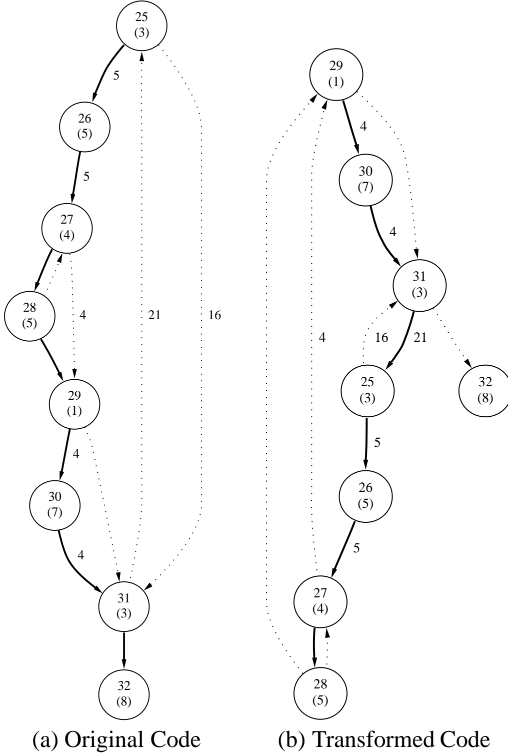
3 Branch Prediction Architectures

Most branch architectures that do not include a BTB incur a misfetch penalty while a branch is decoded. Some architectures use branch delay slots or other mechanisms [5, 7, 17] to avoid this penalty. In this paper we assume the fall-through instruction is fetched while a branch is decoded (except for the branch target buffer architecture). Thus, 'taken' branches always incur a misfetch penalty. We modeled three static branch prediction architectures and two dynamic prediction architectures.

Static Branch Prediction Architectures: The "FALLTHROUGH" model assumes the fall-through execution path is always executed. The "BT/FNT" (backward-taken, forward not taken) assumes backward branches are always taken while forward branches are not taken. This branch model is fairly common, and variants of it are implemented on the HP PA-RISC and the Alpha AXP-21064. The "LIKELY" model assumes that encoded information in the branch instruction indicates whether the branch is likely to be taken or not taken. This branch model is used by several architectures including the Tera [2]. The "likely/unlikely" flag can be set either using compile-time estimates [3] or profile information [9]. We use profile information since it is much more accurate and simple to gather with appropriate tools [23].

Program transformation can help these branch prediction architectures reduce misfetch and misprediction delays. In the FALLTHROUGH architecture, the fall-through path should be executed most frequently, both to reduce misfetch and improve prediction. In the BT/FNT architecture, it's useful to have the fall-through be the most common path, but if that's not possible or cost-effective, the branch target should be placed before the conditional branch so a backwards branch is predicted. Since the branch mispredict penalty is larger than the misfetch penalty, it may be better to correctly predict the backwards branch even if this results in a misfetch. In the LIKELY model, the compiler can specify the likely branch outcome, therefore the code transformations can only eliminate misfetch penalties when making the fall-through the most frequently executed path. We should expect that there are more opportunities for optimization with the FALLTHROUGH method than the BT/FNT model because all taken branches will be mispredicted in the FALLTHROUGH method. Likewise, we would expect more optimization opportunities for the BT/FNT model than the LIKELY model since we can only improve the misfetch rate when transforming programs using the LIKELY model.

Figure 1 shows how code transformations can help each static



(a) Original Code (b) Transformed Code

Figure 1: Benefits of code transformation for `elim_lowering` in ESPRESSO. The darkened edges are fall-through and the dotted edges are taken. Nodes represent basic blocks.

prediction model. Figure 1 shows a portion of the control flow graph from the routine `elim_lowering` in the ESPRESSO benchmark. Nodes are labeled with numbers and the number in parenthesis indicates the number of instructions in that basic block. Edges are labeled by frequency of execution. The edge labeled "16" is executed for 16% of all edge transitions in that subroutine. Unlabeled edges are executed less than 1% of the time. Fall-through edges are darkened while "taken" edges are dotted.

In the original code in Figure 1(a) the LIKELY architecture can correctly predict the most likely targets having misfetch penalties on edges $27 \rightarrow 29$, $31 \rightarrow 25$, and $25 \rightarrow 31$. In comparison, the FALLTHROUGH architecture will mispredict edges $27 \rightarrow 29$, $31 \rightarrow 25$, and $25 \rightarrow 31$ since these are all taken branches. The BT/FNT architecture will also mispredict edges $27 \rightarrow 29$ and $25 \rightarrow 31$, but will correctly predict edge $31 \rightarrow 25$ since the target is before the branch instruction resulting in a backwards branch.

The transformed code in Figure 1(b) is an efficient layout in terms of branch costs for each of the static prediction architectures. Since node 25 is now the fall-through of node 31 all of the architectures can correctly predict edge $31 \rightarrow 25$ without any penalty, and since 31 is laid out before 25 the BT/FNT can accurately predict edge $25 \rightarrow 31$ with only a misfetch penalty. Also, since node 29 is laid out before 27 the branch $27 \rightarrow 29$ can be accurately predicted. The transformed program gives an optimal layout for the BT/FNT since it will have the same prediction as the LIKELY architecture. This is also a good layout for the FALLTHROUGH architecture. Though it still suffers by mispredicting edges $27 \rightarrow 29$ and $25 \rightarrow 31$, but it can predict the less likely executed edge $25 \rightarrow 26$. Notice that in the transformed code that there are two taken edges coming out of node 28. Since one of the edges has to be the fall-through, we need to add a unconditional branch to the fall-through which will in turn jump to the correct destination node. The transformed code in Figure 1(b) gives an efficient transformation for each of the static architectures, but in general, a single branch alignment transformation will not always give an optimal alignment for the different architectures.

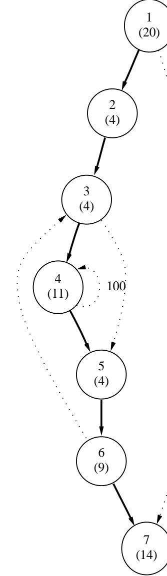


Figure 2: Routine `input_hidden` from ALVINN

Code transformations to reduce branch penalties should consider the underlying branch model when performing the branch alignment. Later, we examine transformation algorithms that model the underlying branch architecture and show that they can perform better than algorithms that do not.

There are many optimizations such as un-rolling loops that we did not investigate. For example when we traced the ALVINN program, which is a neural net simulator, we found that 46% of the time was spent in routine `input_hidden` and another 46% was spent in `hidden_input`. Figure 2 shows the control flow graph for `input_hidden`. Nearly 100% of the branches in that subroutine, or $\approx$ 46% of all branches in ALVINN, arise from a single branch from basic block 4. If we unrolled that loop, duplicating the 11-instruction basic block 4, we could reduce the misfetch penalty for all architectures and improve the branch prediction for the FALLTHROUGH architecture. Normally, loop unrolling is a more complex transformation that also attempts to reduce the total number of executed branches within the unrolled code. We feel that simply duplicating the basic block 4 and then inverting (aligning) the branch condition for the added conditional branches in this example would offer some performance improvement, even if the other optimizations offered by loop unrolling were ignored.

Dynamic Branch Prediction Methods: While static prediction mechanisms, particularly profile-based methods, accurately predict 70-90% of the conditional branches, many current computer architectures use *dynamic prediction*, such as branch target buffers (BTB) and pattern history tables (PHT) to accurately predict 90-95% of the branches.

Originally, BTB's were used as a mechanism for branch prediction, effectively predicting the prior behavior of a branch – even small BTB's were found to be very effective [4, 17, 20, 22, 26]. The Intel Pentium is an example of a current architecture using BTB's – it has a 256-entry BTB organized as a 64 line four-way associative cache. Only branches that are 'taken' are entered into the BTB. If a branch address appears in the BTB, the stored address is used to fetch future instructions. Otherwise, the fall-through address is used. For each BTB entry, the Pentium also uses a two-bit saturating counter to predict the direction of a conditional branch [14].

Conditional branches can be predicted using much simpler mechanisms, but these methods do nothing for misfetch penalties. A *pattern history table* PHT eliminates the site and target addresses from the table and the table only predicts the direction for conditional branches. These designs use the branch site address as an index into a table of *prediction bits*. More recently Pan *et al* [19] and Yeh and Patt [27] investigated *branch-correlation* or *two-level* branch prediction mechanisms. Although there are a number of variants, these mechanisms generally combine the history of several recent branches to predict the outcome of a branch. The simplest example is the *degenerate method* of Pan *et al*. When using a 4096 entry table, the processor maintains a 12-bit shift register (the global history register) that records the outcome of previous branches. If the previous 12 branches that executed were a sequence of three taken branches, six non-taken branches and three more taken branches (TTTNNNNNNTTT), the register might store the value 111000000111_2, or 3591. This value is used as an index into the 4096-entry pattern history table, providing contextual information about particular patterns of branches.

We simulated two PHTs, a direct mapped PHT and the degenerate two-level correlation PHT using a variant that McFarling [16] found to be the most accurate. This method performs an exclusive-or of the branch address with the global history register and uses this as an index into the PHT. Both of the PHTs we simulated contained 4096 2-bit saturating up-down counters, for a total of 1KBytes of storage. We also simulated two BTB configurations. We modeled a 64-entry 2-way associative BTB and a 256-entry 4-way BTB – the latter configuration is used in the Intel Pentium. The BTBs we simulated store only taken branches in the BTB and predict fall-through on a BTB miss. Each BTB entry contains a 2-bit saturating up-down counter used to predict the destination for conditional branches. The BTB in our simulations hold entries for conditional branches, unconditional branches, indirect jumps, procedure calls and returns [5, 26].

4 Branch Alignment Algorithms

We implemented the branch alignment algorithm suggested by Pettis and Hansen [21]. We did not implement the algorithms of McFarling and Hennessy or Bray and Flynn, because they only examine '`if/then/else`' constructs. This limits their effectiveness since many of our transformations are applied to loops. For example, those algorithms would not perform the transformations shown in Figure 1. In our results, we perform branch alignment for each procedure in a program. We are mainly concerned with reducing branch cost, although instruction cache performance may also be improved. We represent a procedure by a directed control flow graph containing a set of basic blocks represented by nodes and edges between nodes. We trace the program execution, recording the number of times each edge is traversed. We call this the *execution weight* for edge e of node n.

Unconditional branch	2	(instruction + misfetch)
Correctly predicted fall-through	1	(instruction)
Correctly predicted taken	2	(instruction + misfetch)
Mispredicted	5	(instruction + mispredict)

Table 1: Cost, in cycles, for different branches.

When transforming a program we look at all nodes that have an out degree of one or two. An unconditional branch is a basic block with a single out-going edge, the 'taken' edge. A conditional basic block has two edges, the 'taken' and the 'fall-through' edges, and a fall through basic block has an out-going 'fall-through' edge. All other edges are given a weight of zero and are not considered when applying branch alignment. Thus, we ignore indirect branches, procedure returns and subroutine calls. In this section we discuss three branch alignment algorithms.

Greedy: Pettis and Hansen [21] proposed two heuristics to align branches. We only describe their bottom-up ('greedy') algorithm, since it has better performance. The Greedy algorithm was directed towards the BT/FNT architecture, and did not consider the implications of different branch architectures. In the terminology of [21], a *chain* is a contiguous sequence of basic blocks threaded by 'head' and 'tail' pointers. The first basic block in each chain has a null head pointer and the last basic block in each chain has a null tail pointer.

The algorithm aligns each procedure in turn. The edge $S \rightarrow D$, where S is the source and D the destination, with the largest weight is selected. The algorithm then attempts to position node D as the "fall-through" of node S. If S does not already have a fall-through basic block, and D does not already have a head, then these two basic blocks are combined into a chain. Otherwise, these blocks cannot be linked. If these basic blocks are part of existing chains, the two chains are merged when the basic blocks are linked. This is repeated until all edges have been examined and chains can no longer be merged. Pettis and Hansen implemented their technique for the HP PA-RISC architecture. This architecture uses the BT/FNT conditional branch prediction model. After all the edges in a procedure have been examined, a precedence relation is defined between chains to determine an ordering between chains that would achieve the best prediction using the BT/FNT model. The chains are positioned using this precedence relation, inserting unconditional branches when needed.

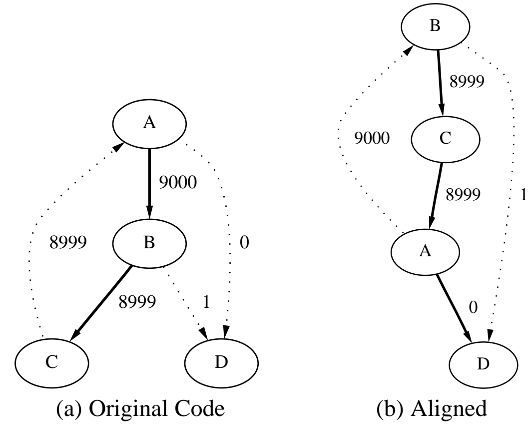
Cost: The Greedy algorithm does not consider the underlining architecture when constructing chains. We include these underlining architecture costs in our algorithms in order to reduce the cost of branches beyond that of the Greedy algorithm. Our architecture assumes specific costs for different branches, shown in Table 1. The "Cost" transformation algorithm tries to minimize the cost of the branches for a procedure using simple heuristics, hoping that each local minimization will result in a global performance improvement.

As in the Greedy algorithm, the Cost algorithm starts with the edge with the highest weight. When we pick an edge $S \rightarrow D$, we determine if having D be the fall-through for S will locally benefit the program using our cost model before trying to link S and D. We examine all the predecessors of D to see if it is more cost effective to connect D to another node. Our algorithm considers basic blocks with one and two exit edges. We consider two possible alignments for single-exit nodes. We examine the cost of aligning the edge as a fall through (thereby avoiding an unconditional branch) or adding an unconditional branch. For example, if S were a single-exit node, we could either include S and D in the same chain, or insert a jump to D at the end of S, allowing S and D to be in different chains. For conditional branches we examine three possible alignments. Assume S has another edge, $S \rightarrow D_2$. We consider including the $S \rightarrow D$ or the $S \rightarrow D_2$ edge in the current chain or adding a jump to the end of S making the jump the fall-through of S. This latter transformation may be useful if it is more cost effective to have both D and D_2 as fall-throughs in other chains.

In certain cases, not aligning either edge of a conditional branch can improve performance on the FALLTHROUGH and BT/FNT architectures. For example, consider a loop consisting of a single basic block, such as that shown in Figure 2. Using the FALLTHROUGH model, the original loop in node 4 incurs a five cycle penalty (one cycle for the branch instruction and four cycles for the misprediction penalty) using our cost-model. It is cost-effective to invert the sense of the conditional ending the block and follow the block with an inserted jump instruction. This combination takes only three cycles (the correctly predicted conditional branch, the unconditional branch and a single misfetch penalty). Any loop can be structured this way - we illustrated the point using the single block loop because the Greedy algorithm would not restructure such loops and they occur frequently.

Try15: Our simulation study showed that the 'Cost' heuristic gave sizable improvements for the FALLTHROUGH architecture, modest improvements for BT/FNT and negligible improvements for LIKELY. We briefly considered using the cost model to assess the cost of every possible basic block alignment using an exhaustive search and selecting the minimal cost ordering. In practice, this sounds expensive, but in the common case procedures contain 5-15 basic blocks. However, most programs have procedures containing hundreds of blocks, making exhaustive search impossible for those procedures. For example, the GCC program contains a procedure (`yyparse`) containing 712 basic blocks. However, many edges were never executed in those large procedures, and a few basic blocks contribute most of the execution time.

We devised a heuristic that balanced time against performance. For each procedure, we select the 15 most frequently executed edges and attempt all possible alignments for these nodes. We then select the next 15 edges, and so on. This allows us to try all possible combinations for each group of 15 nodes. The possibilities to try for each node is similar to that described for the Cost algorithm. For single-exit nodes (unconditional and fall through basic blocks) the two possibilities are to make the outgoing edge either a fall through or taken edge, and for nodes with two-exit edges (conditional branches) we try aligning separately each of the two outgoing edges as the fall-through and then try having neither of the out-going edges as the fall-through. We call this the 'Try15' method. This heuristic took more time than the Greedy and Cost heuristics, but produced better results and still ran in a few minutes. Considering 10 nodes at a time gave slightly worse results than Try15 for a few programs, but it took less than a minute to run and still resulted in better performance than the Greedy algorithm.



(a) Original Code (b) Aligned

Figure 3: Example illustrating where Try15 reduces branch costs. The darkened edges are fall-through and the dotted edges are taken. Nodes represent basic blocks.

To improve the performance of our algorithm we only examined edges that were executed more than once. This eliminated over half of the edges from consideration in each program. If more profiles are used or combined for a program, one could reduce the execution time of the Try15 algorithm by examining only those conditional branches that account for 99% of the executed branches.

For branch alignment algorithms, aligning loops is difficult, and this is one case where our heuristics perform better than the Greedy algorithm. Figure 3(a) shows a fragment of code with a loop. The Greedy algorithm would not modify this code assuming it chooses to align edge $B \rightarrow C$ before it chooses edge $C \rightarrow A$. Whereas, the Try15 algorithm transforms the code as shown in Figure 3(b). Note that in the transformed code the unconditional branch from $C \rightarrow A$ is removed. Using our cost model in Table 1 for the LIKELY and BT/FNT architecture, the execution cost for the original code with the edge-weights shown is $9000 + 8999 + 8999 * 2 + 1 * 5 = 36,002$ cycles, while the cost for our transformed version is $8999 + 9000 * 2 + 1 * 5 = 27,004$ cycles. This reduces the branch execution cost by 33%.

Ideally, we want the most likely path through the loop to be in a single chain. The Greedy and Cost algorithms do not examine enough of the loop to minimize this cost. This is one of the main

reasons why the Try15 heuristic is able to produce better results than the other algorithms. The Try15 heuristic can can try all the combinations to find the correct place to "break" a loop.

5 Experimental Methodology

We constructed two tools to study branch alignment. Initially, we simulated several different branch architectures using trace driven simulation. Later, we implemented the different branch alignment algorithms using the OM [24, 25] system for link-time code transformation. The simulations provide more detailed insight on why branch alignment is useful on the different branch architectures. The implementation illustrates that these techniques have practical value.

During the simulation study, we instrumented the programs from the SPEC92 benchmark suite and other programs, including object-oriented programs written in C++. We used ATOM [23] to instrument the programs. Due to the structure of ATOM, we did not need to record traces and could trace very long-running programs. The programs were compiled on a DEC 3000-400 using the Alpha AXP-21064 processor using either the DEC C compiler or DEC C++ compiler. The systems were running the standard OSF/1 V1.3 operating systems. All programs were compiled with standard optimization (`-O`). We constructed several simulators to analyze the program. Each simulator was run once to collect information about branches targets and a second time to use profile information from the prior run. For the SPEC92 programs, we used the largest input distributed with the SPEC92 suite.

Table 2 shows the basic statistics for the programs we instrumented. The first columns lists the number of instructions traced and the second column gives the percentage of instructions which cause a break in control flow. The columns labeled 'Q-50', 'Q-90', 'Q-99' and 'Q-100' show the number of branch instruction sites that contribute to 50, 90, 99 and 100% of all executed conditional branches in the program. While the next column 'Static' shows the total number of conditional branch sites in each program. Thus, in `doduc`, three branch instructions constitute 50% of all executed branches. The '%Taken' column shows the percentage of conditional branches that are 'taken' during execution. The last five columns provide details about the five types of breaks in control flow encountered during tracing: conditional branches (**CBr**), indirect jumps (**IJ**), unconditional branches (**Br**), procedure calls (**Call**) and procedure returns (**Ret**). Note, that dynamic dispatch calls are implemented as indirect jumps in C++ and are therefore included in the indirect jump metric.

The 'other' programs include: `cfront`, version 3.0.1 of the AT&T C++ language preprocessor written in C++, `groff`, a version of the `ditroff` text formatter written in C++, `idl`, a C++ parser for the CORBA interface description language, `db++`, a version of the 'deltablue' constraint solution system written in C++ and TEX, a text formating system. We selected these programs because we found that the SPEC92 suite did not typify the behavior seen in large programs or C++ programs [6]. For these alternate programs, we used sizable inputs we hoped would exercise a large part of the program – for example, the TEX program formated a 45-page document.

6 Results

For all of our results we only rearrange basic blocks within a procedure, and we do not perform procedure splitting nor any procedure rearranging. Since each branch alignment method adds and removes instructions in the program there is no clear cut performance metric to compare the performance for these different alignments. Simple metrics such as prediction accuracy are not useful, because one method may have removed or added unconditional branches to achieve a particular branch alignment.

We define the *branch execution penalty* (BEP) to be the execution penalty associated with misfetched and mispredicted branches. For our simulations we assumed a misfetched branch causes a one cycle misfetch penalty and a mispredicted branch causes a four cycle mispredict penalty. For the static branch and PHT architectures, unconditional branches, correctly predicted taken conditional branches and direct procedure calls all cause misfetch penalties. Whereas, mispredicted conditional branches, mispredicted returns, and all indirect jumps cause mispredict penalties. Since the BTB architecture tries to predict all types of branches, taken branches (procedure calls, unconditional jumps, and taken conditional branches) found in the BTB do not necessarily cause misfetch penalties. In all of our static and dynamic architecture simulations we simulated a 32-entry return stack [12], which is very accurate at predicting the destination for return instructions.

In order to evaluate the performance of the different alignments and architectures, we add the BEP to the number of instructions executed in the aligned program and divide by the number of instructions executed in the original program. This essentially defines the cycles per instruction relative to the original program. This also assumes that each instruction takes one cycle. For example, if the original program issues 1,000 instructions and encounters 347 cycles from branch penalties, it would have a CPI of 1.347. If a modified program issues 978 instructions, assuming some branches were avoided, and incurred 347 cycles from branch penalties, it would have a relative CPI of $(978 + 347)/1000$, or 1.325 cycles. We call this relative CPI since we are dividing the cost of the aligned program by number of instructions in the original program.

Table 3 shows the relative CPI for each program using the various alignments on the three static branch architectures. The table also shows the percent of executed conditional branches which are fall-through after the alignment has been performed for the varying architectures. The percent of fall-through branches does not change for the Pettis algorithm on the varying branch architectures, whereas the fall-through percentage for the Try15 algorithm changes for each architecture since the cost model algorithm is different for each architecture. Table 4 shows the relative CPI for the PHT and BTB architectures. Arithmetic averages are shown for each group of programs (SPECfp92, SPECint92, and 'Other'). The 'Orig' column for each architecture shows the performance when we instrumented and traced the original program. For the LIKELY architecture, we used the profiles that are used to create the branch alignments in order to predict the likely branch direction for a given branch site. For each architecture, we use the same input to 'align' the program and to measure the improvement from that alignment.

The branch alignment heuristics that use the architectural cost model usually perform better than the simpler Greedy algorithm –

Program	# Insn's Traced	% Breaks	Conditional Branches						Percentage of Breaks during Tracing				
			Q-50	Q-90	Q-99	Q-100	Static	%Taken	**%CBr**	**%IJ**	**%Br**	**%Call**	**%Ret**
alvinn	5,240,969,586	9.09	2	2	102	430	1,622	97.77	98.30	0.02	0.40	0.64	0.64
doduc	1,149,864,756	8.53	3	175	296	1,447	7,073	48.68	81.31	0.01	4.97	6.86	6.86
ear	17,005,801,014	8.10	2	6	32	530	1,846	90.13	61.37	0.05	3.71	17.42	17.46
fpppp	4,333,190,877	2.82	10	51	109	744	6,260	47.74	86.66	0.00	8.01	2.66	2.66
hydro2d	5,682,546,752	6.28	14	74	230	1,613	7,088	73.34	95.84	0.00	1.38	1.39	1.39
mdljsp2	3,343,833,266	10.60	6	14	23	1,010	6,789	83.62	95.43	0.00	4.00	0.29	0.29
nasa7	6,128,388,651	3.08	8	55	277	1,083	6,581	79.29	81.34	0.41	6.34	5.95	5.95
ora	6,036,097,925	7.52	5	11	17	641	5,899	53.24	69.85	0.00	10.65	9.75	9.75
spice	16,148,172,565	12.57	2	38	116	1,762	9,089	71.63	91.56	0.16	3.73	2.28	2.28
su2cor	4,776,762,363	4.36	8	26	60	1,569	7,246	73.07	76.42	0.71	9.02	6.92	6.92
swm256	11,037,397,884	1.65	2	3	13	795	6,080	98.42	99.63	0.07	0.15	0.08	0.08
tomcatv	899,655,317	3.36	3	5	7	515	5,474	99.28	99.86	0.02	0.05	0.03	0.03
wave5	3,554,909,341	5.71	18	82	276	1,331	8,149	61.79	76.68	0.74	5.92	8.33	8.33
compress	92,629,658	13.91	4	12	16	230	1,124	68.25	88.51	0.00	7.59	1.95	1.95
eqntott	1,810,540,418	11.54	2	14	72	466	1,536	90.30	93.47	1.70	1.90	0.70	2.24
espresso	513,008,174	17.11	44	163	470	1,737	4,568	61.90	93.25	0.20	1.88	2.29	2.39
gcc	143,737,915	15.97	245	1,612	3,724	7,640	16,294	59.42	78.85	2.86	5.75	6.04	6.49
li	1,355,059,387	17.67	16	52	127	556	2,428	47.30	63.94	2.24	7.74	12.92	13.16
sc	1,450,134,411	20.93	14	94	336	1,471	4,478	66.88	85.96	0.98	2.62	5.18	5.26
cfront	19,001,390	16.08	112	946	3,055	5,783	15,509	53.18	73.45	2.17	6.40	8.72	9.26
db++	86,457,511	17.56	9	96	173	421	1,639	56.86	54.43	15.04	2.03	6.77	21.73
groff	41,522,284	16.10	86	372	1,021	2,511	7,434	54.17	66.12	4.80	7.80	8.77	12.51
idl	21,138,201	19.61	9	38	154	1,001	3,839	46.70	50.00	12.31	7.55	9.07	21.07
tex	147,827,875	9.99	39	259	790	2,375	6,050	57.46	75.87	2.80	10.06	5.49	5.78

Table 2: Measured attributes of the traced programs.

this is particularly notable in the FALLTHROUGH architecture. The FALLTHROUGH architecture is no longer a realistic architecture to consider, but is used in combination with BTBs – the fall-through can be predicted on a BTB miss. The improved performance occurs because the Try15 heuristic does not align either of the outgoing edges for some conditional branches. Instead, unconditional branches are added to one of the conditional branch edges to take advantage of the FALLTHROUGH prediction cost model. In fact, the Try15 heuristic converts up to 99%, as seen in Table 3, of all conditional branches in some programs to be fall-through in the FALLTHROUGH model. Adding an unconditional jump works especially well for single basic block loops which end with a conditional branch, as described earlier for ALVINN and many of the FORTRAN programs.

The BT/FNT architecture sees reasonable improvement from branch alignment. In the BT/FNT architecture, it is difficult to create chains for the BT/FNT architecture. When forming chains, it is not known where the taken branch will be located in the final procedure until the chains are formed and laid out. The destination of a taken branch could be placed before or after the current node, affecting the final branch prediction costs.

The small benefit for the LIKELY architecture occurs because we eliminate the misfetch penalty for many branches and we can remove unconditional branches from the likely execution path. Eliminating instruction misfetches will be increasingly important as super-scalar architectures become more common – a four-issue super-scalar architecture could encounter a branch every two or three cycles. It should benefit such architectures to have frequent "fall-through" branches. However, the relative CPI metric shown only reflects the improvement of a single issue architecture.

The cost model used for the static architectures is different than that for the dynamic architectures. When examining the costs of aligning a conditional branch for the static architecture, the costs for aligning the conditional branch are clear cut, meaning only one of the targets of the conditional branch can be predicted and the other must always be mispredicted. In the dynamic architectures this is not the case. In order to compensate for the increased accuracy for predicted conditional branches, our cost model for the PHT architectures assume that conditional branches are mispredicted only 10% of the time. Similarly in the BTB architectures we also assume that conditional branches are mispredicted only 10% of the time and in addition, we assume that the BTB architectures have a 10% miss rate. This means that taken unconditional and conditional branches will only cause a misfetch penalty 10% of the time.

As seen in Table 4, branch alignment offers some improvement for the PHT architectures and little improvement to the BTB architectures except for small BTBs. As with the LIKELY architecture, the major improvement in performance for the PHT architecture comes from moving unconditional branches from the frequently executed path and reducing the misfetch penalty that occurs for taken conditional branches. The original program performance for the BTB architecture is already efficient because it stores and predicts indirect jumps, procedure calls, unconditional and conditional branches. The small BTB architecture can benefit more from branch alignment than the larger BTB since only taken branches are stored in the BTB. Therefore removing unconditional branches and making more branches fall-through will cause the aligned program to use less entries in the BTB.

An important observation is that branch alignment reduces the difference in performance between the various branch architectures. For example, the aligned FALLTHROUGH and BT/FNT architectures have almost identical performance. Both are slightly slower than the LIKELY and PHT architectures, while the BTB architecture has the

	Relative Cycles Per Instruction									% of Fall-Through Conditional Branches				
	FALLTHROUGH			BT/FNT			LIKELY					FALLTHROUGH	BT/FNT	LIKELY
Program	Orig	Greedy	Try15	Orig	Greedy	Try15	Orig	Greedy	Try15	Orig	Greedy	Try15	Try15	Try15
alvinn	1.35	1.34	1.17	1.09	1.09	1.09	1.09	1.09	1.09	2.23	3.76	99.57	3.71	3.75
doduc	1.15	1.09	1.05	1.09	1.04	1.04	1.05	1.04	1.03	51.32	68.90	95.08	68.77	92.24
ear	1.20	1.17	1.10	1.08	1.07	1.07	1.08	1.07	1.07	9.87	25.87	92.85	25.80	25.80
fpppp	1.05	1.02	1.02	1.05	1.02	1.02	1.03	1.02	1.01	52.26	84.68	87.38	83.39	83.62
hydro2d	1.18	1.10	1.06	1.10	1.10	1.04	1.06	1.04	1.04	26.66	57.68	95.44	53.43	53.45
mdljsp2	1.34	1.08	1.07	1.30	1.09	1.06	1.13	1.07	1.06	16.38	87.08	90.20	77.79	77.79
nasa7	1.08	1.07	1.04	1.03	1.02	1.02	1.03	1.02	1.02	20.70	26.35	96.84	26.27	26.32
ora	1.13	1.02	1.02	1.12	1.02	1.05	1.05	1.02	1.02	46.76	94.67	94.96	90.36	90.50
spice	1.34	1.29	1.25	1.15	1.14	1.13	1.12	1.11	1.11	28.37	38.37	92.31	37.42	37.74
su2cor	1.11	1.07	1.05	1.05	1.04	1.04	1.04	1.04	1.03	26.93	52.27	89.82	38.12	38.12
swm256	1.06	1.06	1.03	1.02	1.02	1.02	1.02	1.02	1.02	1.58	1.78	99.42	1.76	1.76
tomcatv	1.13	1.08	1.04	1.08	1.02	1.02	1.03	1.02	1.02	0.72	43.71	99.38	43.71	43.71
wave5	1.12	1.09	1.05	1.06	1.04	1.04	1.05	1.04	1.04	38.21	51.27	94.09	50.96	51.04
SPECfp92 Avg	1.17	1.11	1.07	1.09	1.05	1.05	1.06	1.04	1.04	24.77	48.95	94.41	46.27	48.14
compress	1.35	1.14	1.12	1.26	1.17	1.10	1.16	1.12	1.10	31.75	81.73	84.14	68.72	68.72
eqntott	1.40	1.20	1.11	1.26	1.07	1.07	1.12	1.06	1.06	9.70	55.20	97.56	54.89	54.90
espresso	1.40	1.24	1.20	1.28	1.26	1.25	1.19	1.17	1.16	38.10	62.66	84.30	60.97	65.83
gcc	1.34	1.15	1.13	1.30	1.14	1.14	1.17	1.11	1.11	40.57	76.63	87.37	74.79	75.35
li	1.27	1.12	1.11	1.26	1.14	1.13	1.15	1.10	1.10	52.70	83.03	85.63	83.03	83.11
sc	1.51	1.27	1.18	1.36	1.17	1.16	1.20	1.14	1.14	33.12	66.37	90.91	65.66	65.72
SPECint92 Avg	1.38	1.19	1.14	1.29	1.16	1.14	1.16	1.12	1.11	34.32	70.94	88.32	68.01	68.94
cfront	1.25	1.12	1.10	1.23	1.10	1.10	1.13	1.09	1.09	46.82	81.05	89.64	80.52	81.20
db++	1.34	1.22	1.19	1.30	1.18	1.18	1.21	1.17	1.17	43.14	73.96	90.23	73.47	74.35
groff	1.31	1.11	1.10	1.26	1.17	1.09	1.14	1.09	1.08	45.86	84.20	94.06	82.16	84.53
idl	1.31	1.14	1.13	1.30	1.13	1.13	1.19	1.13	1.13	53.30	90.37	96.11	89.96	90.00
tex	1.20	1.10	1.08	1.17	1.09	1.09	1.10	1.07	1.07	42.53	73.23	87.43	70.67	71.43
Other Avg	1.28	1.14	1.12	1.25	1.13	1.12	1.15	1.11	1.11	46.33	80.56	91.49	79.36	80.30

Table 3: Relative cycles per instruction for static prediction architectures and the corresponding % of fall-through branches

	4096 Direct Mapped PHT			4096 Correlation PHT			64-Entry, 2-way BTB			256-Entry, 4-way BTB		
Program	Orig	Greedy	Try15	Orig	Greedy	Try15	Orig	Greedy	Try15	Orig	Greedy	Try15
alvinn	1.09	1.09	1.09	1.09	1.09	1.09	1.01	1.00	1.00	1.00	1.00	1.00
doduc	1.06	1.04	1.04	1.06	1.03	1.04	1.03	1.02	1.02	1.02	1.01	1.01
ear	1.08	1.07	1.07	1.07	1.07	1.07	1.02	1.02	1.02	1.02	1.02	1.02
fpppp	1.02	1.01	1.01	1.02	1.01	1.01	1.01	1.01	1.01	1.01	1.01	1.01
hydro2d	1.05	1.04	1.04	1.05	1.04	1.04	1.01	1.02	1.01	1.01	1.02	1.01
mdljsp2	1.11	1.06	1.05	1.11	1.06	1.05	1.03	1.04	1.03	1.03	1.04	1.03
nasa7	1.03	1.02	1.02	1.03	1.02	1.02	1.00	1.00	1.00	1.00	1.00	1.00
ora	1.05	1.02	1.02	1.05	1.02	1.02	1.01	1.01	1.01	1.01	1.01	1.01
spice	1.12	1.11	1.12	1.11	1.10	1.11	1.04	1.04	1.07	1.04	1.04	1.07
su2cor	1.05	1.04	1.04	1.05	1.04	1.04	1.02	1.02	1.01	1.02	1.02	1.01
swm256	1.02	1.02	1.02	1.02	1.02	1.02	1.00	1.00	1.00	1.00	1.00	1.00
tomcatv	1.03	1.02	1.02	1.03	1.02	1.02	1.00	1.00	1.00	1.00	1.00	1.00
wave5	1.05	1.04	1.04	1.04	1.03	1.03	1.02	1.01	1.01	1.01	1.01	1.01
SPECfp92 Avg	1.06	1.04	1.04	1.06	1.04	1.04	1.02	1.01	1.01	1.01	1.01	1.01
compress	1.15	1.12	1.10	1.15	1.11	1.09	1.08	1.08	1.06	1.06	1.08	1.06
eqntott	1.12	1.06	1.06	1.11	1.06	1.06	1.02	1.01	1.01	1.01	1.01	1.01
espresso	1.17	1.15	1.15	1.14	1.12	1.12	1.11	1.10	1.10	1.07	1.09	1.09
gcc	1.17	1.12	1.12	1.17	1.11	1.11	1.18	1.10	1.10	1.12	1.08	1.08
li	1.15	1.11	1.10	1.12	1.08	1.07	1.13	1.07	1.07	1.07	1.06	1.05
sc	1.17	1.11	1.11	1.16	1.10	1.10	1.08	1.04	1.04	1.04	1.03	1.03
SPECint92 Avg	1.16	1.11	1.11	1.14	1.10	1.09	1.10	1.07	1.06	1.06	1.06	1.05
cfront	1.14	1.09	1.09	1.14	1.09	1.09	1.19	1.09	1.09	1.13	1.07	1.07
db++	1.20	1.17	1.16	1.18	1.15	1.14	1.08	1.04	1.04	1.04	1.03	1.03
groff	1.14	1.09	1.08	1.13	1.08	1.07	1.13	1.07	1.05	1.06	1.05	1.03
idl	1.19	1.13	1.12	1.18	1.12	1.12	1.11	1.02	1.02	1.03	1.01	1.01
tex	1.10	1.07	1.07	1.09	1.06	1.06	1.08	1.05	1.04	1.05	1.04	1.04
Other Avg	1.15	1.11	1.11	1.14	1.10	1.10	1.12	1.05	1.05	1.06	1.04	1.04

Table 4: Relative cycles per instruction for dynamic prediction architectures.

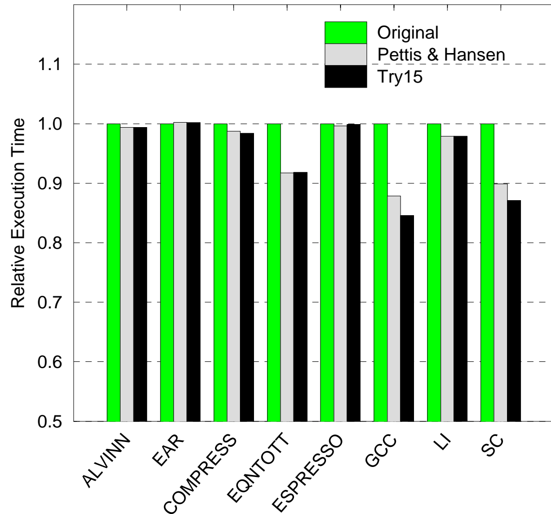


Figure 4: Total execution time improvement on DEC 3000-600 Alpha AXP for the SPEC92 C programs.

best overall performance. In comparing the static BT/FNT architecture performance to the 4096-entry correlated PHT, before alignment the PHT architecture performs 7% better than the BT/FNT architecture, but after alignment it performs only 2% better than the BT/FNT architecture when taking the averaged CPI over all the simulated programs.

Lastly, we note there is a significant difference between the different program classes. The SPECint92 and 'Other' programs see more benefit from branch alignment than the SPECfp92 programs. A reason for this, as seen in Table 2, is that for the SPECfp92 programs $\approx$ 6.5% of the instructions executed cause a break in control flow. Whereas in the the SPECint92 and 'Other' programs $\approx$ 16% of the instructions cause a break in control.

6.1 Performance Comparison

We implemented both the Greedy and Try15 alignment algorithms. Figure 4 indicates the improvement in the total execution time for the SPEC92 C programs on a DEC 3000-600 with an Alpha AXP 21064 processor running OSF/2 V2.0. For each program, we show the execution time for the original program, as compiled by the native compiler, the transformed program using the Pettis and Hanson (Greedy) algorithm, and the transformed program using the Try15 algorithm. We scaled the execution time for each program by the time for the original program.

The programs were compiled as previously described but not linked. We then used OM to link the resulting object files and standard libraries using OM-full as described in [25]. Therefore, the Original program execution times shown in Figure 4 use the standard OM link time optimizations. We then modified OM to produce the desired branch alignments and used this to link the programs.

The Alpha AXP 21064 is a dual issue architecture which uses a combination of dynamic and static branch prediction. Each instruction in the on-chip cache has a single bit indicating the previous branch direction for that instruction. When a cache line is flushed, all the bits are initialized with the bit from each instruction where the sign displacement should be located. Thus the performance expected by this architecture is a cross between a direct mapped PHT table and a BT/FNT architecture.

Not surprisingly the floating point programs, ALVINN and EAR, do not see any benefit from the branch alignment which agrees with our simulation results. We believe some benefit could be gained if the single loop basic blocks (shown in Figure 2) were duplicated. The GCC, EQNTOTT and SC programs benefit the most from branch alignment. It is difficult to understand from where the actual performance improvement from branch alignment comes. Our tools did not allow us to instrument and measure the transformed programs, and our trace simulations did not completely model the Alpha AXP 21064 architecture.

For the simulations described in the previous section, two different chain layout algorithms where used for the Greedy and Try15 alignments. One algorithm laid out chains for a procedure starting with the highest executed chain continuing down to the lowest executed chain. The other algorithm laid out chains using the BT/FNT model described in [21]. We implemented both chain layouts in OM and found that the algorithms that laid the chains out from highest executed to lowest executed performed slightly better than the one that laid out chains using the BT/FNT model. We believe this performance comes from the fact that laying out the chains from highest to lowest executed satisfies many of the branch priorities for the BT/FNT model, and at the same time allowing better cache locality. Therefore the results shown in Figure 4 uses the same Greedy alignment used for all of the simulations (except the BT/FNT simulation) with the highest to lowest chain ordering.

In OM we also implemented the BT/FNT, PHT and BTB alignments for Try15 that were used in the simulations. We found that the BTB alignment performed the same or slightly better than the PHT alignment which was better than the BT/FNT alignment. Recall that when creating a PHT alignment, all taken conditional branches and unconditional branches have a one cycle misfetch penalty associated with it in the cost model. In contrast our BTB cost model assumes a 10% BTB miss rate, which means it assumes the one cycle misfetch penalty only occurs for 10% of the taken branches. In the Alpha AXP 21604 architecture misfetch penalties can be squashed if the pipeline is currently waiting on other stalls. Therefore, the cost model which would more actually fit the Alpha AXP 21604 architecture would assume that taken branches are squashed rufely 30% of the time. The results in Figure 4 uses the same alignment as used for the BTB simulations shown in Table 4.

7 Conclusions

We simulated a number of branch prediction architectures and showed that branch alignment is useful for each architecture. These simulation results assumed a single issue architecture. As wide issue architectures become more popular, branch alignment algorithms will have a larger impact on the performance of programs. When these alignment algorithms were implemented, we saw up to 16% improvement in execution time for the dual issue Alpha

AXP 21604 architecture. The total reduction in program execution time results from a combination of reduction in the misfetch and misprediction penalties, the instruction cache miss rates, and the number of instructions issued.

We described an improved alignment algorithm and also showed that a few branches determine the branch behavior of many common benchmark programs. Our technique addresses a broader class of program structures than [15] and [4] and does not require the recompilation needed by Hwu and Chang[18] or Pettis and Hansen [21].

We have shown how a simple object code transformation, taking no more than a few minutes to run, even for very large programs, can improve a programs performance. Branch alignment does not benefit all programs, but for integer programs a reasonable improvement is seen for the various branch prediction architectures.

Acknowledgments

We'd like to thank Alan Eustace and Amitabh Srivastava for developing ATOM, and especially Amitabh Srivastava for developing OM. We'd also like to thank Keith Farkas, Dennis Lee, and the anonymous reviewers for their useful comments. This work was funded in part by NSF grant No. ASC-9217394, an ARPA Fellowship and a DEC-WRL summer internship. This work is part of a continued effort to make languages such as C++ suitable for scientific computing.

References

[1] W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformation. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[3] T. Ball and J. R. Larus. Branch prediction for free. In *1993 SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 1993.

[4] Brian Bray and M. J. Flynn. Strategies for branch target buffers. In *24th Workshop on Microprogramming and Microarchitecture*, pages 42–49. ACM, ACM, 1991.

[5] Brad Calder and Dirk Grunwald. Fast & accurate instruction fetch and branch prediction. In *21st Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*. ACM, April 1994.

[6] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. Technical Report CU-CS-698, University of Colorado-Boulder, January 1994.

[7] David R. Ditzel and Hubert R. McLellan. Branch folding in the CRISP microprocessor: Reducing branch delay to zero. In *14th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 2–9. ACM, ACM, June 1987.

[8] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, 1974.

[9] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, Mass., October 1992. ACM.

[10] S. J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Transactions on Software Engineering*, 14(11):1640–1644, 1988.

[11] D. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM Systems Journal*, 10(3):168–192, 1971.

[12] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th Annual International Symposium of Computer Architecture*, pages 34–42. ACM, May 1991.

[13] Brian Kernighan. Optimal sequential partitions of graphs. *Journal of the ACM*, 18(1):34–40, 1971.

[14] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 21(7):6–22, January 1984.

[15] Scott McFarling. Program optimization for instruction caches. In *3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1988.

[16] Scott McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[17] Scott McFarling and John Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. ACM, 1986.

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

[19] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Mass., October 1992. ACM.

[20] Chris Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

[21] Karl Pettis and Robert 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.

[22] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*. ACM, 1981.

[23] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, 1994.

[24] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, March 1993.

[25] Amitabh Srivastava and David W. Wall. Link-time optimizations of address calculation on a 64-bit architecture. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, 1994.

[26] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *25th Annual International Symposium on Microarchitecture*, pages 129–139, Portland, Or, December 1992. ACM.

[27] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium of Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.