Branch Vanguard: Decomposing Branch Functionality into Prediction and Resolution Instructions

Daniel S. McFarlin
Carnegie Mellon University
dmcfarlin@cmu.edu

Craig Zilles
University of Illinois at Urbana-Champaign
zilles@illinois.edu

Abstract

While control speculation is highly effective for generating good schedules in out-of-order processors, it is less effective for in-order processors because compilers have trouble scheduling in the presence of unbiased branches, even when those branches are highly predictable. In this paper, we demonstrate a novel architectural branch decomposition that separates the prediction and deconvergence point of a branch from its resolution, which enables the compiler to profitably schedule across predictable, but unbiased branches. We show that the hardware support for this branch architecture is a trivial extension of existing systems and describe a simple code transformation for exploiting this architectural support. As architectural changes are required, this technique is most compelling for a dynamic binary translation-based system like Project Denver.

We evaluate the performance improvements enabled by this transformation for several in-order configurations across the SPEC 2006 benchmark suites. We show that our technique produces a Geomean speedup of 11% for SPEC 2006 Integer, with speedups as large as 35%. As floating point benchmarks contain fewer unbiased, but predictable branches, our Geomean speedup on SPEC 2006 FP is 7%, with a maximum speedup of 26%.

1. Introduction

Control dependence is a major performance challenge for **in-order** machines, especially those that have issue width greater than one, independent of whether they are superscalar, VLIW, DSP, or GPU. While some machines will issue post-branch instructions in parallel with branch instructions (*e.g.*, the DEC Alpha 21164 [11]), the benefit of doing so is limited, as the primary impact of control dependence is on compiler generated code schedules [20, 31]. That is, *even with perfect branch prediction*, control dependence impacts performance on in-order machines. Scheduling across basic block boundaries carries two challenges: hoisted instructions must not have user visible side-effects that violate the semantics of the source program, and the benefit of hoisting instructions must be balanced with the cost of unnecessarily executing instructions hoisted from the wrong path.

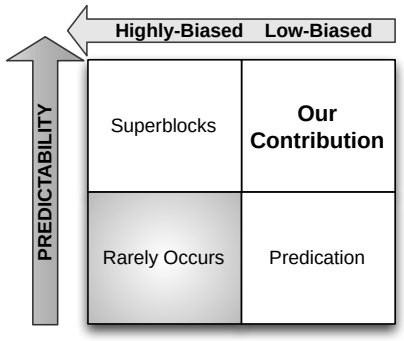


Figure 1: Proposed Transformations for Conditional Non-Loop Branches based on Bias and Predictability

To extract greater ILP, schedulers for in-order machines attempt to build wider issue groups by hoisting instructions from a sequence of successive basic blocks. The sequence of basic blocks selected from which to hoist for most of the well-known techniques (*e.g.*, trace scheduling [14, 6, 37], superblocks [21], hyperblocks [30]) is based on the **bias** of the branches. The formation of these aggregate scheduling regions is typically founded on the presence of highly-biased branches (*i.e.*, branches that tend to resolve in one-direction substantially more frequently than the other direction). Integer code features many such branches including branches that are so heavily biased that they can be assert converted out of existence with little cost [34].

The presence of **unbiased** branches poses a problem for compilers. The classic solution has been predication [1]. Predication, in some form or another, is a common feature in many modern architectures and is particularly useful for **unpredictable** hammocks. In these situations, the cost of converting the control dependence into a data dependence and executing both paths is less than the amortized cost of the branch mispredictions that are being removed. If, however, the branch is predictable, these costs (of fetching/issuing instruc-

ISCA'15, June 13-17, 2015, Portland, OR USA
 ISBN 978-1-4503-3402-0/15/06..$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750400

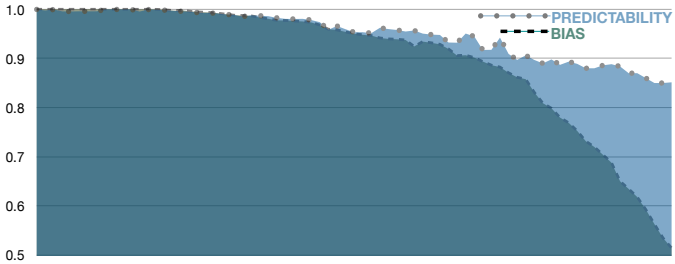


Figure 2: Integer Predictability Remains High vs. Bias for the Top 75 Conditional Forward Branches Averaged Across SPEC 2006 Int

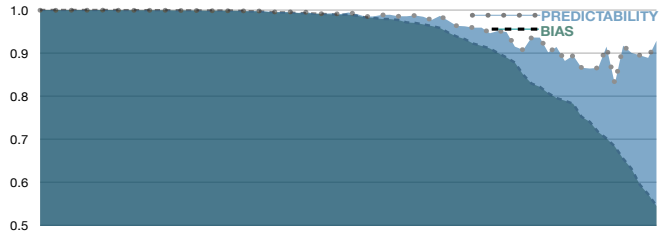


Figure 3: Floating Point Predictability Remains High vs. Bias for the Top 75 Conditional Forward Branches Averaged Across SPEC 2006 FP

tions with false predicates) are being incurred merely because the compiler doesn't have a better way to express the code.

This suitability of control speculation mechanisms can be summarized as shown in Figure 1. When branches are highly biased, superblock formation is effective. As branch predictability is almost universally higher than branch bias, these branches are typically all predictable. Predication is most profitable for unbiased, unpredictable branches, which leaves the region of **predictable, but unbiased branches**, which is the focus of this paper.

In most programs, there are a non-trivial number of branches whose predictability significantly exceeds their bias. Figures 2 and 3 graph the predictability and bias of the top 75 most-executed forward branches (sorted by bias) averaged across SPEC 2006 Int and FP, respectively. For the first half of the graph, predictability and bias remain high and track each other so well they are virtually indistinguishable, but toward the end of the graphs, branch bias begins to diverge from branch predictability. As predictability declines, bias declines much more rapidly. One way to view this graph is that roughly one third (SPEC INT) to one half (SPEC FP) of the time that a branch goes against its preferred direction, the processor would correctly predict that. This discrepancy between branch predictability and bias currently represents a lost opportunity for code scheduling on in-order machines, and this discrepancy **only increases** as branch predictor accuracy improves.

This paper makes the following contributions:

- We propose a novel architectural decomposition of branches into a branch prediction instruction and a branch resolution instruction.
- We describe a low-complexity transformation, the Decomposed Branch Transformation, that enables in-order superscalars to exploit exposed branch predictability.
- We examine the modest hardware support needed for this transformation.
- We evaluate the performance improvements enabled by this transformation for several in-order configurations across the SPEC 2000 and SPEC 2006 benchmark suites
- We show that for SPEC 2000 and SPEC 2006 Integer, our technique produces a geometric mean speedup of 11% with maximum speedups of 35% and 18%, respectively. For SPEC 2000 and SPEC 2006 Floating Point, our technique produces a geometric mean speedup of 7% with maximum speedups of 26% and 20%, respectively.
- We demonstrate our technique's sensitivity to branch predictor accuracy.

The remainder of this paper is organized as follows: Section 2 explores the decomposition of branch behavior which enables the transformation described in full detail in Section 3. Section 4 describes the front-end and back-end hardware required to support this transformation. In Section 5, we show and analyze the performance improvement our transformation can achieve on different in-order superscalar configurations. Section 6 examines the other side-effects of our transformation particularly the impact of increased code size. Related Work is discussed in Section 7 and we conclude in Section 8.

2. Decomposing Branch Behavior

It has previously been recognized by many that branches (as encoded by most architectures) perform many functions in a single operation [38, 41, 16, 29, 24, 43]. Previously published decompositions typically recognize three components of branch functionality:

- Target specification (TS): provide the information necessary to construct the branch target.
- Condition computation (CC): a comparison operation on one or more program values to determine if the branch should be taken or not taken.
- Control flow transfer (CFT): the program point at which the control flow divergence actually takes place.

Figure 4 shows how this branch functionality has been previously decomposed into multiple operations. Subfigure 4(a) shows the implementation commonplace in modern architectures where all of the functionality is performed by a single operation; note we make no distinction between architectures that use condition codes vs. register operands because in both cases the branch indicates how to interpret those program values.

Subfigure 4(b) shows the separation of the branch target computation from the rest of the branch. Since the target of a direct branch is known statically, this information (often with a static branch prediction) can be provided to a machine well before the branch instruction to allow the processor to prefetch/fetch the instructions at the branch target [41, 39]

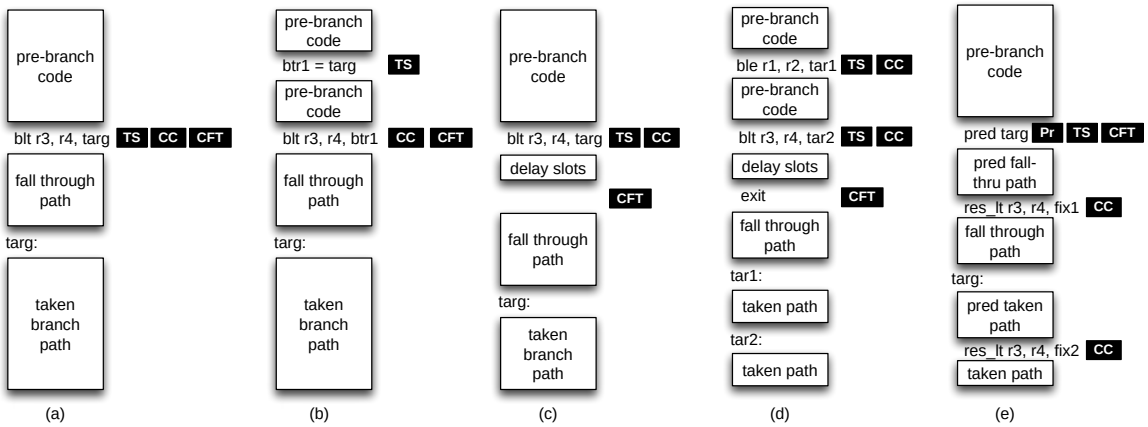


Figure 4: Decomposing branch functionality: TS: Target Specification, CC: Condition Computation, CFT: Control Flow Transfer, and Pr: Prediction. (a) in a traditional branch architecture, (b) early specification of a branch target (and storage in a branch target register(btr)) permits prefetching, (c) branch delay slots implicitly encode the control flow transfer after a number of instructions executed whether the branch is taken or not, (d) explicitly marking the point of the control flow transfer permits using it for multiple branches, and (e) our proposed branch decomposition hoists the control flow transfer with respect to the condition computation.

and/or load them into a special fetch buffer [8] without the need for a branch target buffer.

Subfigures 4(c) and (d) shows the separation of the control flow transfer from the rest of the branch functionality. This decomposition is incredibly effective when the branch condition can be computed early (*e.g.*, in some numerical codes where the control flow is data independent) because it obviates the need to predict the branch, provided that the branch condition can be computed sufficiently early. The formulation shown in Subfigure 4(c) is typically referred to as *branch delay slots* with the location of the control flow transfer specified implicitly by the branch. The number of delay slots can either be fixed architecturally [18] or encoded in the branch instruction [12]. Subfigure 4(d) shows the IBM ACS branch formulation where the location for the control flow transfer is indicated using an `exit` instruction [38], which additionally allows multiple branch instructions to be associated with a single exit instruction to implement multi-way branches as shown; the exit instruction transfers control to the target of the first branch instruction to compute a true predicate.

In these previous decompositions, the control flow transfer remains the final step of the branch, and the benefit comes from hoisting other parts of the branch functionality above this step. As such, it is important to note that for many programs it is difficult to significantly hoist branch resolution computations due to data dependences, especially for superscalar machines. It is for exactly this reason that our proposed formulation hoists the control flow transfer *above* the branch resolution.

2.1. Prediction-Resolution Decomposition

Our proposed decomposition recognizes these previous three parts of a branch functionality, but also considers a fourth component that is of equal importance.

- Prediction (Pr): the point at which the front end predicts the branch's outcome.

We consider prediction explicitly, because in our proposed branch decomposition, we hoist the control flow transfer *above* the branch resolution, which means we fundamentally need a prediction to decide which way the branch should go. Specifically, we decompose a branch into two instructions:

1. **a predict instruction**: this operation consists only of a `predict` opcode and a target. When fetched, this operation is predicted by the branch predictor and, if predicted taken, fetch continues at the target PC.
2. **a resolve instruction**: this instruction looks like a normal branch, but uses a special opcode (*e.g.*, `resolve_lt` for `blt`). It is always predicted not-taken by the front end. If the branch resolves contrary to the prediction from the `predict` instruction, control is transferred to the target of the resolve instruction. In any case, the branch predictor entries associated with the `predict` instruction are updated.

This decomposition is shown in Subfigure 4(e). Note that, because the predict operation includes the control flow divergence, statically there are two resolve instructions associated with each predict instruction, one each for the predict taken and predict not-taken paths. In the Section 3, we show how this instruction is used by the compiler.

2.2. Architectural Change via DBT

As our proposal represents a non-trivial architectural change in the specification of a relatively fundamental operation (*e.g.*, branches) and the technique is targeted at addressing a problem found only in in-order microarchitectures, we're not proposing that this should be adopted for main stream ISAs. Instead, the motivation for this technique is the class of dynamic binary translation (DBT)-based architectures like the Transmeta Crusoe [9] and Nvidia's Project Denver [7]. In these architectures, a mainstreams ISA like x86 or ARM, is translated

by software at run-time to a microarchitecture-specific (often VLIW) hidden ISA. As all code generation for this hidden ISA is controlled by the microprocessor vendor, the hidden ISA can be re-defined each generation to add and remove architectural features. In such a context, there is little cost to extending the hidden ISA and ideas can be evaluated based on their hardware/software cost relative to the performance they provide.

These VLIW-based DBT architectures typically provide many features for facilitating compiler scheduling. Many of these are complementary to the decomposed branch transformation described next, as our transformation typically enables additional opportunities for traditional optimizations. Specifically, three kinds of general mechanisms are useful:

1. **Non-faulting or deferred-faulting load instructions** enable hoisting loads (to overlap their latency with other operations) onto paths where they wouldn't have been executed in the original program. A variety of mechanisms have been proposed and implemented to permit the suppression and handling of faults from control speculation [32, 22].
2. **Support for data speculation** enables hoisting loads past may-aliasing stores, for which a variety of mechanisms exist. Notably, in the context of DBT architectures, it has been shown that data misspeculation can be made to be rare [9], so that recovery code can be placed on separate pages of memory, so as to not impact the instruction cache behavior of the program.
3. **Additional registers to hold speculative values** enables the compiler to hoist more aggressively without spilling to the stack.

We assume the above support as described in Section 5.

3. The Decomposed Branch Transformation

Figure 5 shows a high-level view of the general code transformation we perform to decompose a predictable (90% on both paths) but low-biased (60/40) branch's prediction point from its resolution point. We present this transformation as a sequence of correctness preserving mini-transformations shown in each subfigure.

The first Subfigure, 5(a), shows a conditional branch contained in a block labeled **A** with two possible successor blocks, **B** and **C**. The conditional branch consists of a compare (`cmp`) followed by a branch (`br`) instruction.

Subfigure 5(b) decomposes the `br` instruction into a `predict` instruction and an `resolve` instruction, as discussed in Section 2.1. In splitting the branch, we simultaneously split the block **A** into 3 blocks: the original block **A** and two new blocks **A′** that contain only the `resolve` instructions. The leftmost block **A′** is the path that is predicted to continue to block **B**, while the rightmost is predicted to continue to block **C**. Since the prediction could be incorrect, both **A′**s provide paths to both blocks **B** and **C**.

Through this transformation, we create a pair of highly-biased branches, because the branches in the **A′** blocks are only taken the 10% of the time when the original branch was mispredicted. These highly-biased branches create new profitable opportunities for code scheduling, as we demonstrate. For example, Subfigure 5(c) shows how the compare instruction and other instructions from block **A** can be pushed down into the **A′**s, because the predict instruction is not data dependent on any of the instructions in **A**.

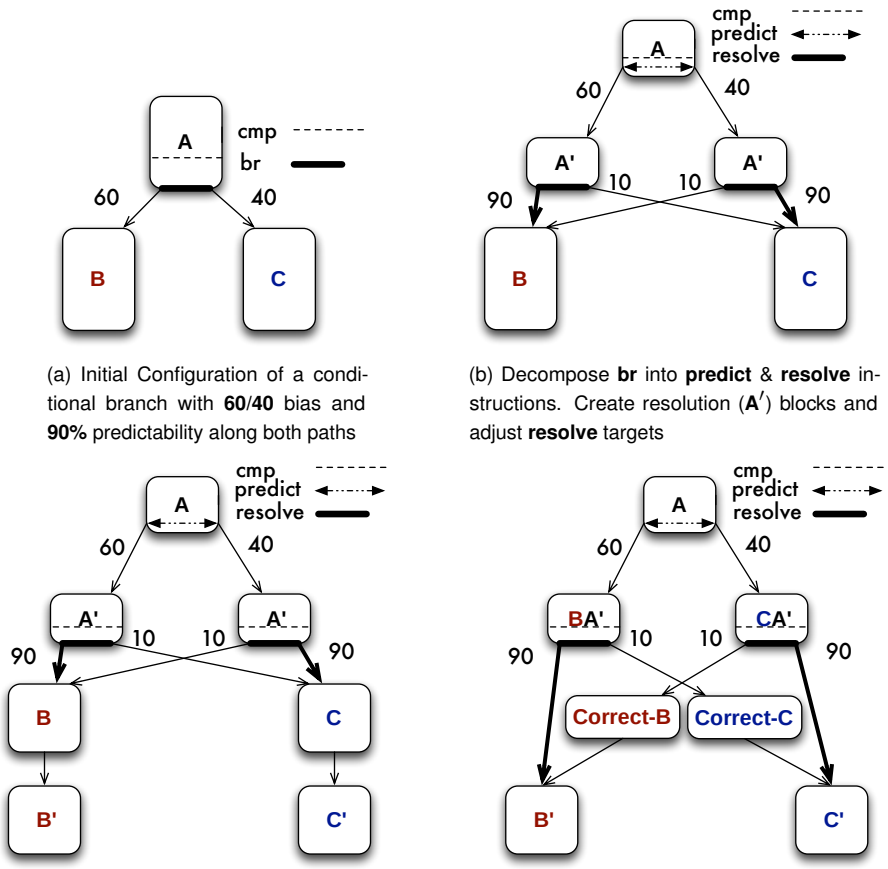
Subfigure 5(d) is largely akin to well-known Superblock formation with associated compensation code; we split the **B** and **C** blocks into two portions; the upper portions of both blocks consist of instructions that would be profitable to hoist into block **A′**; we subsequently hoist these instructions into the new **A′** blocks. By doing so, we can overlap the pushed down contents from block **A** with the hoisted contents of blocks **B** and **C**. When this hoisting is complete, the `resolve` instructions for **BA′** and **CA′** need to fall-thru to **B′** and **C′**, respectively.

Since we typically restrict our transformation to branches with good predictability, most of the time the branch is resolved successfully and execution falls-thru to the **B′** or **C′** block. Should the `resolve` detect a misprediction, we need to redirect control to the correct path. The remaining **C** and **B** blocks (now labelled **Correct-C** and **Correct-B**) serve as compensation code before jumping back into the main flow in blocks **C′** and **B′**, respectively.

A concrete example of a code segment which has high predictability and low bias (from SPEC 2006 omnetpp's `cArray::add(cObject*)` function and simplified for clarity) is shown and subsequently transformed in Figure 6. The major benefit of the transformation for this code segment is overlapping the load latency of the loads in block **A** with the loads in blocks **B** and **C**, which were serialized by the branch in the original code. Saving a load latency is significant on high-frequency machines with multi-cycle cache hits.

Due to space limitations, we compress steps while providing an alternate perspective on the transformation; rather than hoisting instructions from the successor blocks into the resolution blocks we can conceptualize the transformation as pushing the branch resolution slice down into both successor blocks. Subfigure 6(b) shows how the branch resolution has been pushed down both paths; note the presence of lines `2` and `3` in both the **BA′** and **CA′** blocks. Line `3` is the instruction that determines if the branch was resolved in the predicted direction; block **A** now ends with line `3p` which represents the point where the front-end performs the branch prediction. The `loads` shown in lines `5` and `7` in **BA′** and line `40` in **CA′** are distinguished by a `+` sign to indicate to the back-end to suppress faults resulting from their speculative execution.

The new location of line `6` in **B′** and line `41` in **C′** shows that `stores` have been pushed down below the branch resolution point in both blocks. There is little benefit to speculatively executing these stores and had we retained them in their original position, they might have corrupted state in the event of a branch misprediction.



(a) Initial Configuration of a conditional branch with **60/40** bias and **90%** predictability along both paths

(b) Decompose **br** into **predict** & **resolve** instructions. Create resolution (**A′**) blocks and adjust **resolve** targets

(c) Split **B** and **C** blocks into upper (hoistable) and lower portions. Push **cmp** instruction into **resolve** blocks

(d) Hoist upper **B** and **C** blocks into resolution blocks. Add correction code as needed

Figure 5: General Decomposed Branch Transformation

The low register-pressure of the surrounding context for this code segment obviates the need for temporary registers. More generally, we may need to write to temporary registers in the speculative portions (**BA′** and **CA′**) to prevent the clobbering of live-in values for the alternate path in the event of a branch misprediction. Often, we can hide the moves from these temporaries back into architected registers in the shadow of the resolution instruction, where the write-back of the moves can be suppressed in the event of misprediction.

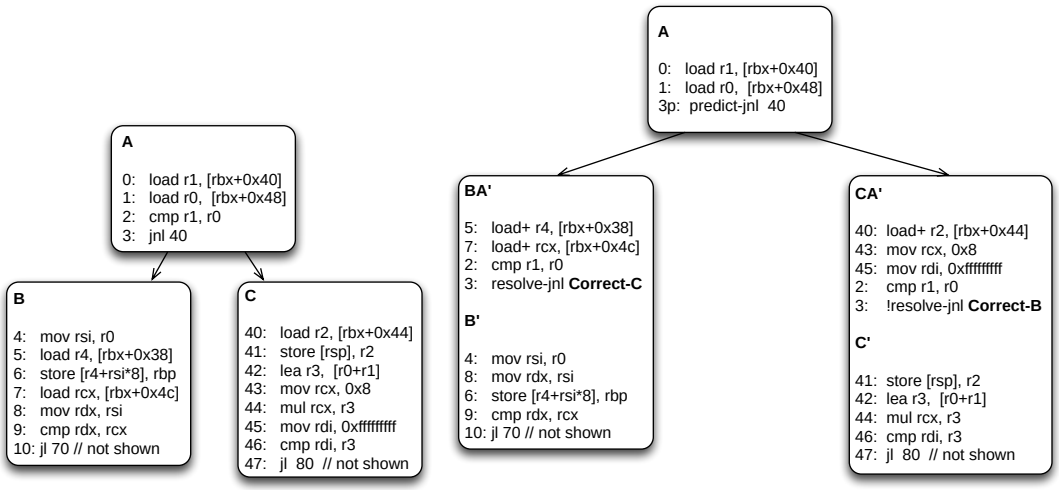
Subfigure 6(c) shows correction code blocks which, in this instance, merely duplicate the hoisted instructions. More generally, arbitrary compensation code can be included in these blocks to undo the effects of executing wrong-path instructions so as to avoid needing to store speculative values in temporary registers. The **Correct-B, Correct-C** blocks directly update architectural state since both correction blocks are guaranteed to be on the correct path.

4. Hardware Support

Our decomposed branches require support in the front end to correctly update the branch predictor. The speculative compilation opportunities created by the Decomposed Branch Transformation benefit from additional architectural support, as has been previously researched. We discuss this hardware support presently.

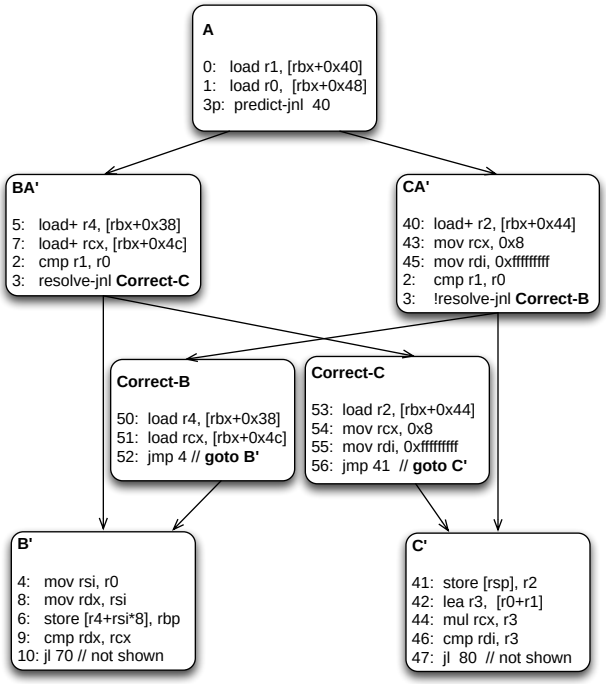
Decomposing branch prediction from branch condition evaluation (they are separate instructions in our machine) slightly complicates the way in which the branch predictors are trained and the manner in which the machine recovers from branch mispredictions. Specifically, our branch resolution instruction (the `resolve` instruction) serves two purposes: 1) it redirects fetch (when our original prediction was incorrect) to fix-up code that patches up state so that we can resume execution along the correct path, and 2) it provides feedback to update the predictor associated with the prediction instruction. Since the PC of the prediction instruction and the PC of the branch resolution instruction are no longer the same, we need to re-associate the outcome of the branch with the prediction of the branch. We maintain this association with a small buffer in the front-end called the Decomposed Branch Buffer (DBB).

The task of maintaining this buffer is somewhat simplified by the fact that branches in our machine are fetched and executed in-order and the fact that the compiler does not reorder or interleave `prediction` and `resolve` instructions. Collectively, these properties enable us to use a simple circular buffer where decomposed branch meta-data is maintained in FIFO order via a tail pointer. Figure 7 shows the placement of the DBB, its integration into the front-end of the machine, and



(a) Initial Code

(b) Portion of the branch resolution slice pushed down both paths. Stores pushed below branch resolution



(c) Addition of correction blocks and new branch targets

Figure 6: Decomposed Branch Transformation For omnetpp

the key operations associated with the DBB's upkeep.

Subfigure 7(a) shows an insert operation on the DBB; a decomposed prediction instruction is detected after decode and is represented by the shaded instruction in the Fetch Buffer. Like any other conditional branch, the prediction instruction goes through the branch predictor. The resulting prediction is piped back to the fetch-unit as normal. In addition, we increment the DBB tail pointer and write the prediction and all of the metadata necessary for a predictor update (*e.g.*., indices) into the DBB at the location indexed by the DBB tail pointer. The prediction instruction is then dropped from the Fetch Buffer since its main function, steering the fetch unit, has been fulfilled.

At some point in the future, the corresponding branch resolution instruction is fetched (shown as the striped instruction in the Fetch Buffer in Subfigure 7(b). This resolution instruction always corresponds to the previous (in program order) prediction instruction, which is the one referenced by the DBB tail pointer. The value of the DBB tail pointer register is then read out of the register and stored with the branch resolution instruction as it moves down the pipeline.

The DBB requires minor modifications to the re-steer logic's control path and data path which we show in Subfigure 7(c). Should a resolution instruction detect a misprediction, the correct path PC is piped back to the fetch unit along with the index into the DBB that was inserted into the resolution

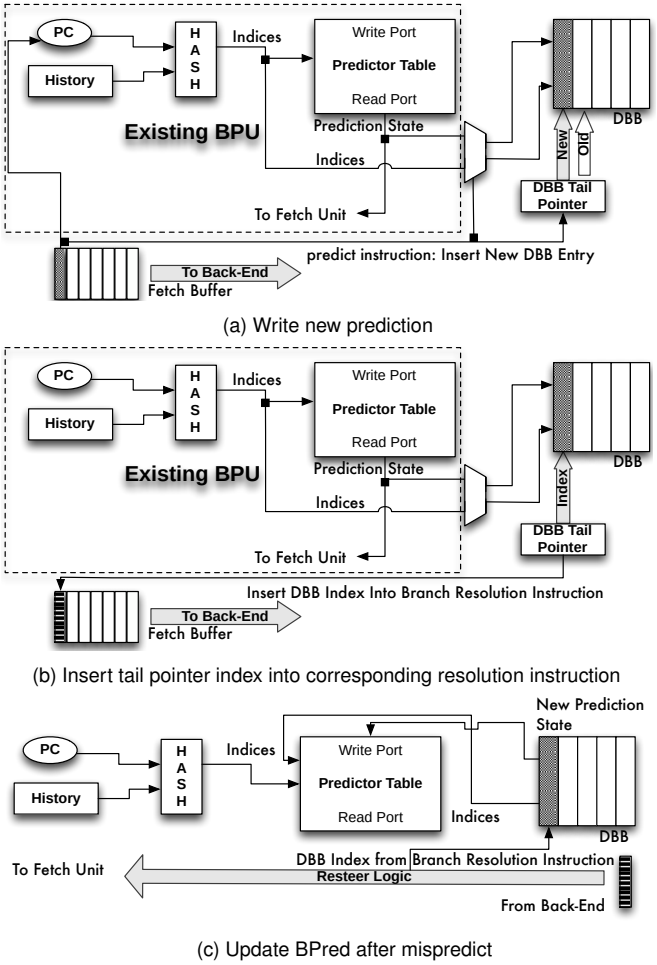


Figure 7: The Decomposed Branch Buffer and its operations. The area shaded in grey denotes existing HW structures and data/control paths

instruction when it passed through the front-end. This index is used to access the DBB entry that contains the prediction meta-data and the indices into the predictor. We then update the predictor with the new predictor metadata that is a function of the metadata stored in the table and the correct path data. In this way, the mispredictions detected by the resolution instruction are associated with the prediction instruction. In the event of a non-decomposed branch misprediction, the DBB tail pointer needs to be recovered to maintain correct correspondence between prediction and resolution instructions; the same mechanism used to recover branch history can be used for this purpose.

While the compiler doesn't re-order or interleave prediction and resolution instructions, exception control flow resulting from interrupts, exceptions, and context switching can potentially do so. There are two approaches to handle these occurrences. First, we could ignore them, as they are relatively uncommon and only result in spurious branch predictor updates. By construction, the DBB always associates a resolution instruction with the immediately previous prediction instruction, so the machine recovers from any mismatch each time a prediction instruction is fetched. If the frequency of these spurious updates is deemed too high, a second approach would be to mark the DBB entries as invalid on one of these exceptional control flow events and suppress any branch predictor updates using an invalid DBB entry.

We size the DBB empirically. In practice, the number of outstanding decomposed branches is small; there tends to be significant back-pressure in the in-order due to head-of-line blocking in the issue stage. We found that 16 entries were more than sufficient, resulting in a 4-bit DBB index that needs to be stored with resolution instructions. In our implementation, each DBB entry contains 24 bits: 16 bits for the indices into the branch prediction table hierarchy and 8 bits for the prediction metadata. The DBB itself requires one read and one write port.

Key Structures	Configuration Parameters
Bpred	PTLSim default: GShare, 24 KB 3-table direction predictor, 4K-entry BTB, 64-entry RAS
Front-End	5 stages, **Experimentally Varied 2/4/8 wide Fetch/Decode/Dispatch**, 32-entry FetchBuffer
Execution Ports	**Experimentally Varied 2/4/8**
Functional Units	Up to 2 x LD/ST, 2 x INT/SIMD-Permute, 4 x 64-bit SIMD/FP, 1-cycle bypass
L1 Caches	8-way 32 KB L1-D$, 4-way 32 KB L1-I$, 64B lines, 4-cycle latency
L2 Cache	16-way 256KB Unified, 12-cycle latency
L3 Cache	32-way 4MB LLC, 25-cycle latency
Miss Handling	64-entry Miss Buffer, 64-entry Load Fill Request Queue
Main Memory	140-cycle latency

Table 1: Machine Configuration Parameters

5. Experimental Evaluation

In our evaluation, we use LLVM 3.5, PTLSim [44], and the SPEC 2006 and 2000 benchmark suites. As it is not profitable to decompose all branches, we use a profile-guided strategy to select the set of branches to transform. We run the **TRAIN** input sets to completion in PTLSim to collect branch bias and predictability. We transform forward branches[1] whose predictability exceeds bias by at least 5%; this heuristic provided the best overall performance. Code generation is handled by LLVM at the -O3 level with PGO and SSE2 vectorization.

For performance evaluation, we use **REF** input sets. We select hot spots for simulation using Intel's Software Development Emulator [5]; simulation is up to 10B dynamic instructions. We simulate three different width in-order superscalars (2-wide, 4-wide, and 8-wide, with configuration parameters shown in Table 1). For our experimental system, PTLSim was extended with a decoupled branch buffer (DBB, as described in Section 4) as well as the support for speculative compilation as discussed in Section 2.2, including "shadow registers" which allows the compiler to re-use architectural registers for

[1] Backwards-taken branches i.e. loop branches tend to be highly biased and highly predictable and are ably handled by well-known loop transformations such as modulo scheduling.

speculative computations and commit them when resolution instructions commit.

Name	SPD	PBC	PDIH	ALPBB	ASPCB	PHI	MPPKI	PISCS
h264ref	23.1	50.2	11.8	9.6	21.6	76.9	6.7	15.6
perlbench	18.4	45.1	12.7	4.9	23.0	50.5	1.6	14.8
astar	16.3	40.3	14.6	6.6	21.51	64.4	13.6	10.2
omnetpp	12.2	23.0	8.1	2.5	79.8	80.3	5.4	12.1
xalancbmk	12.1	24.7	5.0	1.7	27.5	72.4	7.3	9.6
sjeng	10.3	25.6	7.8	3.2	27.7	60.0	12.8	10.6
gobmk	9.1	14.4	5.6	3.4	23.1	84.1	17.8	9.6
gcc	9.1	23.6	6.8	2.3	29.5	68.7	8.4	10.0
mcf	8.1	32.6	6.1	6.0	107.2	73.8	25.5	6.8
bzip2	7.7	13.7	3.5	3.4	26.3	61.3	6.5	9.8
hmmer	6.0	10.3	3.7	12.2	32.5	97.8	1.2	9.5
libquantum	3.1	10.7	5.4	0.8	127.3	78.1	1.1	10.4
wrf	26.3	22.2	14.9	6.1	34.2	69.0	0.5	10.2
povray	22.3	26.5	8.6	3.0	22.7	84.8	2.6	9.7
tonto	11.1	29.3	9.2	3.1	17.1	79.8	4.4	8.3
gamess	11.0	44.1	11.4	3.5	23.4	54.0	4.4	14.6
calculix	10.4	19.2	4.14	2.1	23.7	10.2	7.7	10.1
milc	7.7	23.5	12.8	10.1	32.8	76.9	1.3	10.0
soplex	7.2	13.1	4.3	1.0	37.5	48.7	5.5	9.7
namd	7.0	23.2	5.6	2.4	24.9	94.2	2.1	10.3
lbm	6.6	28.6	16.6	19.5	55.6	66.1	0.2	8.8
gromacs	6.2	21.8	2.4	4.1	38.9	88.3	2.8	10.4
sphinx3	4.4	16.4	2.4	2.6	39.9	86.6	4.9	9.9
bwaves	3.3	27.9	12.3	9.2	25.3	8.8	2.7	11.5
GemsFDTD	3.0	9.4	2.6	3.2	35.5	67.8	1.3	10.4
zeusmp	2.3	21.7	3.6	14.7	40.0	84.9	0.6	11.3
dealII	2.1	11.0	0.8	2.5	24.3	10.9	3.5	8.1
cactusADM	1.4	11.2	0.2	35.3	23.6	97.1	0.5	10.1
leslie3d	1.0	9.4	1.0	32.7	46.0	94.2	0.4	10.7

Table 2: SPEC 2006 Int and FP Metrics Sorted by Speedup.
- **SPD: % Speedup (Geomean) Over All REF inputs for 4-wide**
- **PBC: % of Static Forward Branches Converted**
- **PDIH: Avg. % of Dynamic Instructions Hoisted Above Converted Branch**
- **ALPBB: Avg. Number of Loads Per Basic Block**
- **ASPCB: Avg. Stall Cycles Per Converted Branch**
- **PHI: Avg. % of Instructions Hoistable From Succeeding Basic Block**
- **MPPKI: Branch Mispredictions per Thousand Instructions**
- **PISCS: % Increase in Static Code Size**

5.1. Integer Performance Improvement Analysis

Figures 8 through 11 show the performance improvement on integer code from SPEC 2000 and SPEC 2006. Since branch bias varies, sometimes significantly, depending on the reference input we break out the best performing reference input for each benchmark for both suites in Figures 9 and 11.

We find that performance improvements are largely correlated to the following factors (quantified in Table 2):

- The number of forward branches that exhibit relatively higher predictability than bias (**PBC** guided in-part by **MPPKI**)
- the number of independent instructions, especially loads (i.e. MLP), in the successor blocks that we can hoist (a function of **ALPBB**, **PDIH** and **PHI**).
- tendency to stall at branch resolution (**ASPCB**)
- modest L1 D$ miss rate as we do not employ Runahead [10, 4] or iCFP [19]
- good I$ performance

Since the last two factors have been extensively analyzed for SPEC (particularly in [23]) we omit them from the table. We provide further analysis in the text below.

On balance, the 4-wide configuration tends to benefit the most from our approach; the transformation can balance the 4-wide's functional unit utilization to a greater degree than the narrow 2-wide configuration, while we can rarely fully utilize the 8-wide.

perlbench, h264ref and *astar* cluster together speedup wise at the very high end: *perlbench* has a large number of candidate branches most with very good predictability, a low D$ miss-rate and above average MLP. *h264ref* is similar to *perlbench* with respect to L1 D$ behavior and predictability but exposes more MLP. Even though *astar* is not highly predictable (averaging around 85%), its unbiased branches are relatively highly predictable and the majority of those branches meet our selection criteria. The first reference input for *astar* also boasts a very low L1 D$ miss rate (around 2%); we see a lower average performance improvement due to the second reference input's higher (more than 5%) L1 D$ miss rate.

A second performance class consists of *omnetpp, xalancbmk* and *sjeng*. Despite having comparatively high L1 D$ miss-rate, both *omentpp* and *xalancbmk* benefit from having ample MLP in the successor blocks for the candidate branches. *sjeng* is comparable in terms of candidate branches, has very low L1 D$ and I$ miss-rates but has a noticeably higher MPPKI which somewhat reduces performance.

The next performance class consists of *gobmk, gcc* and *mcf*. *gobmk* and *mcf* both suffer from high MPPKI which inhibits performance uplift; *mcf* also has a large number of long latency misses which is difficult for code generator to cover with useful instructions. In a similar vein, *gcc* has a much higher than average ASCPB coupled with low PHI and low ALPBB which makes it challenging to cover the branch resolution stalls.

At the low-end of the performance spectrum are *bzip2, hmmer* and *libquantum*. *bzip2* has slightly more eligible branches than the other two. *hmmer*, despite being highly predictable and very well behaved from a D$ perspective is dominated by a large loop that contains very few candidate forward branches. *libquantum* is in some respects more like an FP benchmark and despite being highly predictable contains few candidate forward branches and little in the way of hoistable loads.

The SPEC 2000 Integer suite tends to be more predictable and better behaved L1 D$ and I$-wise than its successor as reflected in the higher Geomean performance improvements shown in Figures 10 and 11. For benchmarks present in both SPEC 2006 and SPEC 2000, the relative performance improvements are decidedly mixed. For example, *mcf* in SPEC 2000, has significantly better performance than its SPEC 2006 counterpart due the former's significantly higher branch predictability; the same observation applies to *gcc*. In contrast, while the SPEC 2000 versions of *bzip2* and *perl* have lower MPPKI than the SPEC 2006 versions the SPEC 2006 versions

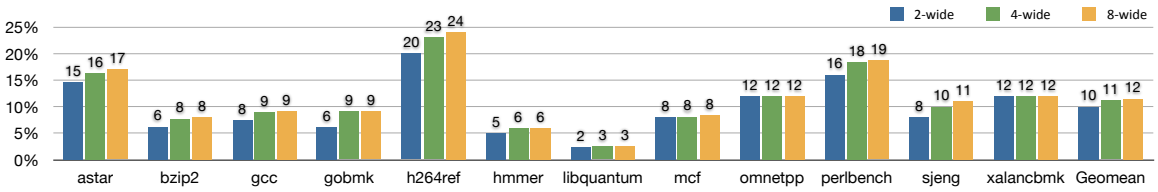


Figure 8: SPEC 2006 Integer Performance: % Speedup Over Baseline Averaged Over ALL Reference Inputs

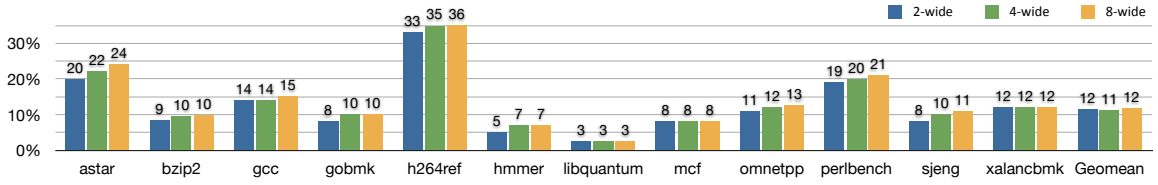


Figure 9: SPEC 2006 Integer Performance: % Speedup Over Baseline for Top Performing Reference Input

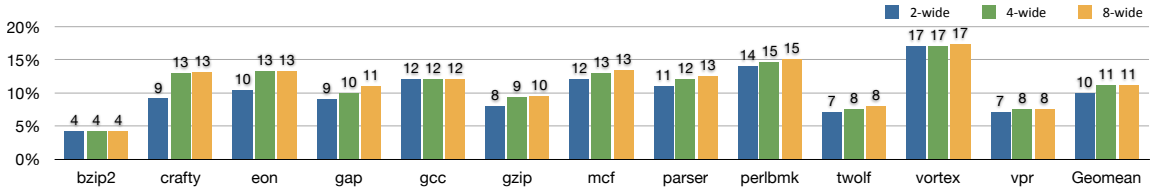


Figure 10: SPEC 2000 Integer Performance: % Speedup Over Baseline Averaged Over ALL Reference Inputs

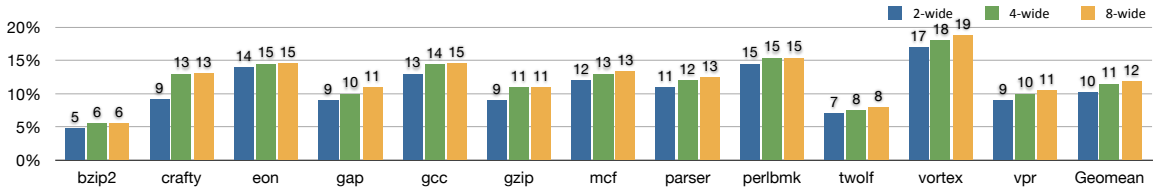


Figure 11: SPEC 2000 Integer Performance: % Speedup Over Baseline for Top Performing Reference Input

contain a greater number of unbiased branches which meet our selection criteria.

crafty, eon, gap, parser and especially *vortex* all have a relatively large number of eligible forward branches (greater than 22% on average), very good predictability, ample MLP and L1 D$ miss rates which are typically 2.5% or lower. *gzip* has comparable PBC but a significantly higher (2-3x) average L1 D$ miss rate. The lowest performing benchmarks, *twolf* and *vpr*, have few (around 11% PBC) eligible forward branches combined with higher than average L1 D$ miss rates.

5.2. Floating Point Performance Improvement Analysis

Figures 12 and 13 show the performance improvement attained on the SPEC 2006 and SPEC 2000 Floating Point (FP) benchmark suites respectively. FP benchmarks tend to be highly predictable, have low I$/ D$ miss-rates, and good MLP. However, these benchmarks also feature quite biased forward branches and large basic blocks, which often permit the branch resolution to be scheduled in such a way as to avoid stalling the in-order. These latter characteristics tend to limit the applicability of our technique as reflected in the lower Geomean speedups relative to Integer. Nevertheless, the very high predictability of FP benchmarks enables some noticeable performance improvement when there are a large number of forward branch candidates.

Referring to Figure 12, the top performers: *wrf, povray, tonto, gamess, calculix, soplex* and *milc* have, on average, greater than 29% of their forward branches eligible for transformation with low MPPKI, and low ASPCB. The next class of performers, *namd, lbm* and *gromacs* have, on average, only 20% of their forward branches eligible for transformation with an average branch predictability of 95% and mid single digit PDIH; performance uplift for *lbm* is inhibited by a high ASPCB *GemsFDTD, bwaves, sphinx3*, and *zeusmp* continue the downtrend in performance with fewer than 18% of their forward branches eligible and low single digit PDIH. The remaining benchmarks in SPEC 2006 have either very low PBC or very few hoistable instructions (PDIH).

Figure 13 shows the performance improvement for SPEC 2000 FP. Overall, this suite contain benchmarks which have fewer eligible forward branches than SPEC 2006. Consider the top performers: *art, ammp* and *mesa*. As we would expect for FP, these benchmarks have high predictability, averaging over 97%. However, these top performers have only, on av-

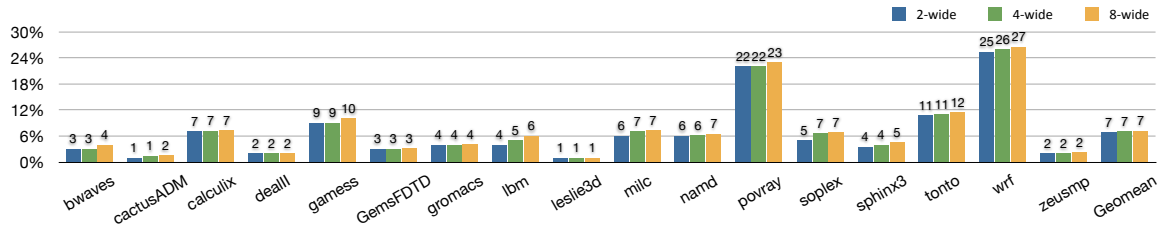


Figure 12: SPEC 2006 Floating Point Performance: % Speedup Over Baseline Averaged Over ALL Reference Inputs

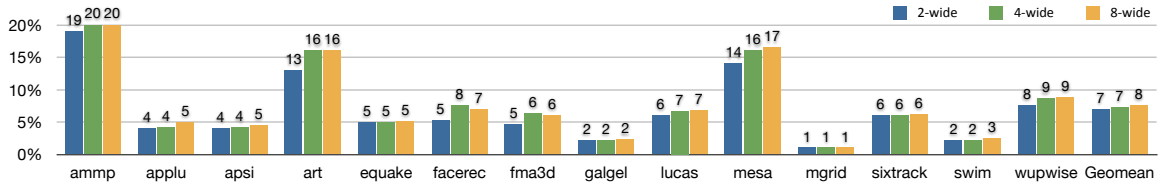


Figure 13: SPEC 2000 Floating Point Performance: % Speedup Over Baseline Averaged Over ALL Reference Inputs

erage, less than 20% of their forward branches eligible for transformation. The falloff in performance improvement is much more pronounced for SPEC 2000 than SPEC 2006; *wupwise* and *facerec*, the next in overall performance, average only 15% eligible forward branches. The remaining benchmarks exhibit little performance improvement due to only 10%, on average, eligible forward branches.

5.3. Branch Predictor Sensitivity

The present work uses a *gshare*-derived conditional branch predictor which is of modest complexity and represents a reasonable trade-off between size and prediction accuracy for an in-order machine. Since the benefit of our technique improves with increased branch predictor accuracy, this conservative choice of branch predictors pessimizes our results. To explore the sensitivity of our performance improvement to prediction accuracy, we simulated a series of ever improving conditional branch predictors, culminating in a 64-KB version of ISL-TAGE described in [35]. We find that changing the branch predictor primarily impacts four hard-to-predict integer benchmarks `astar, sjeng, gobmk, mcf`. In these programs, the speedup from our technique **improves** (over the baseline with the improved branch predictor) roughly 0.3% for each 1% reduction in misprediction rate.

6. Transformation Side-Effects

Our transformation necessarily increases code size and will typically increase the number of wrong-path instructions issued. We examine the impact of these side-effects presently.

6.1. Code Size and Its Impact on Performance

Prior work has shown that SPEC 2000 and SPEC 2006 are remarkably well behaved from an I$ perspective: miss-rates for a 32KB I$ average 0.24% and 0.33% respectively [23]. The degree to which the I$ miss rate actually impacts total performance is a complicated matter and depends on the criticality of those instructions whose fetch and subsequent processing were delayed by the I$ miss. To greatly simplify, absent the miss, would the instructions contained in the miss have even been ready to execute once they entered the window and more importantly, were these instructions on the critical path? If not, then the I$ miss has minimal impact on performance; the backend is otherwise fully engaged issuing the ancestors of the instructions contained in the I$ miss [13].

It is particularly important to distinguish the impact that I$ misses have on the performance of out-of-orders (OOO) vs. in-orders; the opportunity cost for the OOO is potentially greater as the OOO may have had the backend resources available to execute the younger instructions. In contrast, the in-order is very unlikely to have had available issue slots due to the frequent head-of-line blocking that in-orders experience [19]; our 4-wide, in-order shows virtually **no** overall performance degradation (less than 0.5% Geomean for SPEC2k6) when the 32KB I$ capacity is reduced by 25% to 24KB. The average static code size increase for our transformation (as shown in the **PISCS** column in Table 2) is around 9% which is comparable to the average code size difference we have observed when using ICC vs. LLVM.

One potential area of concern is the conjunction of branch mispredictions and I$ misses, that is to say an I$ miss that occurs when the front-end attempts to fetch the correct path after a branch misprediction. Such a conjunction will almost certainly degrade performance for an in-order; OOO architectures may be able to overlap the execution of right path instructions prior to the mispredicted branch and the front-end redirect. In fact, I$ misses and branch mispredictions may both be symptoms of a large instruction working-set that causes aliasing in the branch predictor. Fortunately, the probability of an I$ miss coinciding with a branch misprediction is low; we observed that around 15% of I$ misses occur under a branch misprediction when running the baseline code on a 4-wide in-order. For our experimental configuration, both the I$ miss rate and the portion of I$ misses that occur under a branch misprediction increased by less than 1% on average.

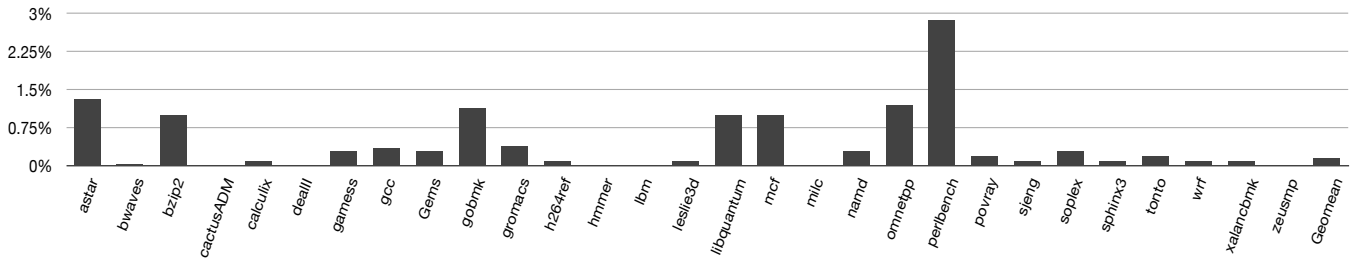


Figure 14: % Increase in Instructions Issued For 4-wide Experimental Configuration Vs. 4-wide Baseline

6.2. Efficiency

Our transformation must take care in targeting forward branches with comparatively low predictability, particularly those where we can form large basic blocks prior to the branch resolution; minimizing the number of wrong path instructions that the machine issues is critical to preserving the in-order's overall energy efficiency. While both our baseline and experimental configurations can issue instructions in the shadow of a conditional branch, the experimental configuration's ability to hoist these instructions before the branch resolution can potentially result in a non-trivial increase in the number of wrong-path instruction issued. As suggested by Figures 2 and 3, a number of unbiased forward branches do in fact have comparatively low predictability. Fortunately, in these cases, the size of the basic blocks we can form tends to be small particularly in the case of less predictable integer code.

Figure 14 demonstrates the overall efficiency (as measured by the increase in issued instructions) of our transformation for the entire SPEC 2006 benchmark suite; SPEC 2006 is generally less predictable than its predecessor and therefore a good indicator of our overall efficiency. For floating point benchmarks, overall predictability is very high which results in a negligible increase in issued instructions. The increase in the number of instructions issued for integer code is larger but generally quite small on average (under 1%).

7. Related Work

A considerable amount of early work was devoted to replacing, minimizing or supplementing the perceived high area, power and complexity of hardware dynamic branch predictors, in addition to the mechanisms described in Section 2. One proposed architecture chose not to employ branch prediction and implemented the equivalent of variable branch delay slots by separating branch resolution and control-flow change instructions into separate queues [3]. Another hardware/software approach dynamically or statically partitions branch slices from program slices into separate instruction streams which are executed on dedicated cores [40, 33].

A recent work, "Control-Flow Decoupling" [36], focused on code restructuring combined with specific HW support to reduce branch mispredictions with an emphasis on hard-to-predict loop branches; our technique focuses on exploiting predictable (but unbiased) branches to generate better code. The two techniques are orthogonal, and in fact "Control-Flow Decoupling" could be used as a means to increase the number of predictable, but unbiased branches and hence the speedup from our technique.

Another avenue of research investigated the degree to which hardware branch prediction could be replaced with software by having the compiler synthesize a sequence of instructions that used current register values to predict future branch outcomes [29, 2]. Profile-based code transformations have been developed to enhance static branch prediction schemes as an alternative to history-based HW implementations [42].

Conversely, computer architecture researchers attempted to co-opt compiler techniques (like predication) in hardware. Dynamic predication in OOO without dedicated predication support in the ISA was proposed [28]. Subsequent work examined the tension between static and dynamic predication in the OOO with an increasing emphasis on the use compile-time and run-time profiling to guide static and dynamic code generation [27, 26, 25].

A recent processor, IBM's Cell BE, used prepare-to-branch and static branch prediction in the streaming processors (SPEs) which operated on domain-specific workloads for which these features are well suited [17, 15].

8. Conclusion

In-order superscalar processors are challenged by control-dependencies coupled with code generators that can only aggregate basic blocks based on branch bias rather than branch predictability. This is a major limitation as branch predictability is often significantly higher than branch bias. In this paper, we describe a low-complexity, low-overhead, simple transformation, the decomposed branch transformation, that decomposes the prediction point of an unbiased but predictable forward branch from the branch resolution point. This transformation provides a code generator with the ability to break control dependencies, cover load-to-use latencies and find valuable memory-level-parallelism. We showed how this transformation has little impact on I$ miss-rates, requires very modest front-end and back-end hardware support and benefits both integer and floating point benchmarks across the SPEC 2000 and SPEC 2006 benchmark suites.

References

[1] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83. New York, NY, USA: ACM, 1983, pp. 177–189. [Online]. Available: http://doi.acm.org/10.1145/567067.567085

[2] D. I. August, D. A. Connors, J. C. Gyllenhaal, and W.-m. W. Hwu, "Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results," in *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture*, ser. HPCA '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 84–. [Online]. Available: http://dl.acm.org/citation.cfm?id=548716.822702

[3] E. Brunvand, "The nsr processor," in *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, vol. i, Jan 1993, pp. 428–435 vol.1.

[4] H. W. Cain and P. Nagpurkar, "Runahead execution vs. conventional data prefetching in the ibm power6 microprocessor," in *ISPASS*, 2010, pp. 203–212.

[5] M. Charney, "Intel software development emulator." [Online]. Available: https://software.intel.com/en-us/articles/pintool

[6] R. P. Colwell, R. P. Nix, J. J. O. Donnell, D. B. Papworth, and P. K. Rodman, "A vliw architecture for a trace scheduling compiler," in *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1987, pp. 180–192.

[7] B. Dally, ""project denver"processor to usher in a new era of computing," Jan. 2011. [Online]. Available: http://blogs.nvidia.com/blog/2011/01/05/project-denver-processor-to-usher-in-new-era-of-computing

[8] J. W. Davidson and D. B. Whalley, "Reducing the cost of branches by using registers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: ACM, 1990, pp. 182–191. [Online]. Available: http://doi.acm.org/10.1145/325164.325138

[9] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges," in *Proceedings of the International Symposium on Code Generation and Optimization*, 2003, pp. 15–24.

[10] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th International Conference on Supercomputing*, ser. ICS '97. New York, NY, USA: ACM, 1997, pp. 68–75. [Online]. Available: http://doi.acm.org/10.1145/263580.263597

[11] J. Edmondson, P. Rubinfeld, R. Preston, and V. Rajagopalan, "Superscalar instruction execution in the 21164 alpha microprocessor," *Micro, IEEE*, vol. 15, no. 2, pp. 33–43, Apr 1995.

[12] M. Farrens and A. Pleszhun, "Implementation of the pipe processor," *Computer*, vol. 24, no. 1, pp. 65–70, Jan 1991.

[13] B. A. Fields, S. Rubin, and R. Bodik, "Focusing processor policies via Critical-Path prediction," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, Jul. 2001, pp. 74–85. [Online]. Available: http://www.cs.wisc.edu/~bodik/research/isca01a.pdf

[14] J. A. Fisher, "Trace scheduling: a technique for global microcode compaction," vol. 30(7), pp. 478–490, 1981.

[15] J. Fritts and W. Wolf, "Evaluation of static and dynamic scheduling for media processors," in *Proceedings of the 2nd Workshop on Media Processors and DSPs*, ser. Micro '00, 2000.

[16] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, "Pipe: A vlsi decoupled architecture," *SIGARCH Comput. Archit. News*, vol. 13, no. 3, pp. 20–27, Jun. 1985. [Online]. Available: http://doi.acm.org/10.1145/327070.327117

[17] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in cell's multicore architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1109/MM.2006.41

[18] J. Hennessy, N. Jouppi, F. Baskett, T. Gross, and J. Gill, "Hardware/software tradeoffs for increased performance," in *Proceedings of the First International Symposium on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS I. New York, NY, USA: ACM, 1982, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/800050.801820

[19] A. Hilton, S. Nagarakatte, and A. Roth, "icfp: Tolerating all-level cache misses in in-order processors," *IEEE Micro*, vol. 30, no. 1, pp. 12–19, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.20

[20] P. Y. T. Hsu and E. S. Davidson, "Highly concurrent scalar processing," in *Proceedings of the 13th Annual International Symposium on Computer Architecture*, ser. ISCA '86. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 386–395. [Online]. Available: http://dl.acm.org/citation.cfm?id=17407.17401

[21] W. M. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. O. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *Journal of Supercomputing*, vol. 7, no. 1, pp. 229–248, Mar 1993. [Online]. Available: http://www.crhc.uiuc.edu/IMPACT/ftp/journal/jsc.superblock.93.pdf

[22] Intel, "Intel itanium processor 9500 series refence manual. software development and optimization guide," Intel Technical Manual, 2012.

[23] A. Jaleel, "Memory characterization of workloads using instrumentation-driven simulation: A pin-based memory characterization of the spec cpu2000 and spec cpu2006 benchmark suites." [Online]. Available: http://www.jaleels.org/ajaleel/workload/SPECanalysis.pdf

[24] V. Kathail, M. Schlansker, and B. Rau, "HPL PlayDoh architecture specification: Version 1.0," Hewlett-Packard Laboratories, Tech. Rep. HPL-93-80, Feb. 1993.

[25] H. Kim, J. Joao, O. Mutlu, and Y. N. Patt, "Profile-assisted compiler support for dynamic predication in diverge-merge processors," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 367–378. [Online]. Available: http://dx.doi.org/10.1109/CGO.2007.31

[26] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Diverge-merge processor: Generalized and energy-efficient dynamic predication," *IEEE Micro*, vol. 27, no. 1, pp. 94–104, Jan. 2007. [Online]. Available: http://dx.doi.org/10.1109/MM.2007.9

[27] H. Kim, O. Mutlu, J. Stark, and Y. Patt, "Wish branches: combining conditional branching and predication for adaptive predicated execution," in *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, Nov 2005, pp. 12 pp.–54.

[28] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *Parallel Architectures and Compilation Techniques, 1998. Proceedings. 1998 International Conference on*, Oct 1998, pp. 278–285.

[29] S. Mahlke and B. Natarajan, "Compiler synthesized dynamic branch prediction," in *Microarchitecture, 1996. MICRO-29.Proceedings of the 29th Annual IEEE/ACM International Symposium on*, Dec 1996, pp. 153–164.

[30] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *In Proceedings of the 25th International Symposium on Microarchitecture*, 1992, pp. 45–54.

[31] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: Is it speculation or dynamism?" in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451143

[32] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 44–55, Mar. 2003. [Online]. Available: http://dx.doi.org/10.1109/MM.2003.1196114

[33] A. S. Nadkarni and A. Tyagi, "A trace based evaluation of speculative branch decoupling," in *Computer Design, 2000. Proceedings. 2000 International Conference on*. IEEE, 2000, pp. 300–307.

[34] N. Neelakantam, R. Rajwar, S. Srinivas, U. Srinivasan, and C. Zilles, "Hardware atomicity for reliable software speculation," in *Proceedings of the 34th International Symposium on Computer Architecture*, 2007, pp. 174–185.

[35] A. Seznec, "A new case for the tage branch predictor," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 117–127. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155635

[36] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-flow decoupling," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 329–340. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2012.38

[37] G. Shobaki, K. Wilken, and M. Heffernan, "Optimal trace scheduling using enumeration," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 4, pp. 19:1–19:32, Mar. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498690.1498694

[38] M. Smotherman, "Documentation project for the IBM ACS-1 Supercomputer," Jun. 2010. [Online]. Available: http://www.cs.clemson.edu/~mark/acs.html

[39] A. Srivastava and A. Despain, "Prophetic branches: a branch architecture for code compaction and efficient execution," in *Microarchitecture, 1993., Proceedings of the 26th Annual International Symposium on*, Dec 1993, pp. 94–99.

[40] A. Tyagi, H.-C. Ng, and P. Mohapatra, "Dynamic branch decoupled architecture," in *Computer Design, 1999.(ICCD'99) International Conference on*. IEEE, 1999, pp. 442–450.

[41] W. J. Watson, "The ti asc: A highly modular and flexible super computer architecture," in *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part I*, ser. AFIPS '72 (Fall, part I). New York, NY, USA: ACM, 1972, pp. 221–228. [Online]. Available: http://doi.acm.org/10.1145/1479992.1480022

[42] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VI. New York, NY, USA: ACM, 1994, pp. 232–241. [Online]. Available: http://doi.acm.org/10.1145/195473.195549

[43] H. C. Young, "Code scheduling methods for some architectural features in pipe," *Microprocessing and Microprogramming*, vol. 22, no. 1, pp. 39 – 63, 1988. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0165607488900063

[44] M. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*, April 2007, pp. 23–34.