Control-Flow Decoupling

Rami Sheikh, James Tuck, Eric Rotenberg
Department of Electrical and Computer Engineering
North Carolina State University
{rmalshei, jtuck, ericro}@ncsu.edu

Abstract

Mobile and PC/server class processor companies continue to roll out flagship core microarchitectures that are faster than their predecessors. Meanwhile placing more cores on a chip coupled with constant supply voltage puts per-core energy consumption at a premium. Hence, the challenge is to find future microarchitecture optimizations that not only increase performance but also conserve energy. Eliminating branch mispredictions – which waste both time and energy – is valuable in this respect.

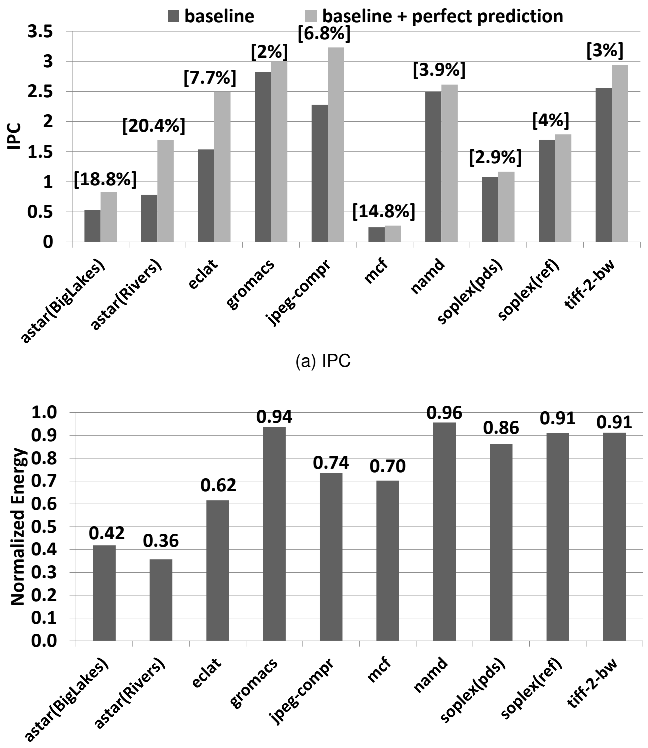
We first explore the control-flow landscape by characterizing mispredictions in four benchmark suites. We find that a third of mispredictions-per-1K-instructions (MPKI) come from what we call separable *branches: branches with large control-dependent regions (not suitable for if-conversion), whose backward slices do not depend on their control-dependent instructions or have only a short dependence. We propose control-flow decoupling (CFD) to eradicate mispredictions of separable branches. The idea is to separate the loop containing the branch into two loops: the first contains only the branch's predicate computation and the second contains the branch and its control-dependent instructions. The first loop communicates branch outcomes to the second loop through an architectural queue. Microarchitecturally, the queue resides in the fetch unit to drive timely, non-speculative fetching or skipping of successive dynamic instances of the control-dependent region.*

Either the programmer or compiler can transform a loop for CFD, and we evaluate both. On a microarchitecture configured similar to Intel's Sandy Bridge core, CFD increases performance by up to 43%, and reduces energy consumption by up to 41%. Moreover, for some applications, CFD is a necessary catalyst for future complexity-effective large-window architectures to tolerate memory latency.

1. Introduction

Good single-thread performance is important for both serial and parallel applications, and provides a degree of independence from fickle parallelism. This is why, even as the number of cores in a multi-core processor scales, processor companies continue to roll out flagship core microarchitectures that are faster than their predecessors. Meanwhile placing more cores on a chip coupled with stalled supply voltage scaling puts per-core energy consumption at a premium. Thus, the challenge is to find future microarchitecture optimizations that not only increase performance but also conserve energy.

Eliminating branch mispredictions is valuable in this respect. Mispredictions waste both time and energy, firstly, by fetching and executing wrong-path instructions and, secondly, by repairing state before resuming on the correct path. Figure 1a shows instructions-per-cycle (IPC) for several applications with hard-to-predict branches. The first bar is for our baseline core (refer to Table 3 in Section 5) with a state-of-art branch predictor (ISL-TAGE [28, 29]) and the second bar is for the same core with perfect branch prediction. Each application's branch misprediction rate is shown above its bars. Speedups with perfect branch prediction range from 1.05 to 2.16. Perfect branch prediction reduces energy consumption by 4% to 64% compared to real branch prediction (Figure 1b).



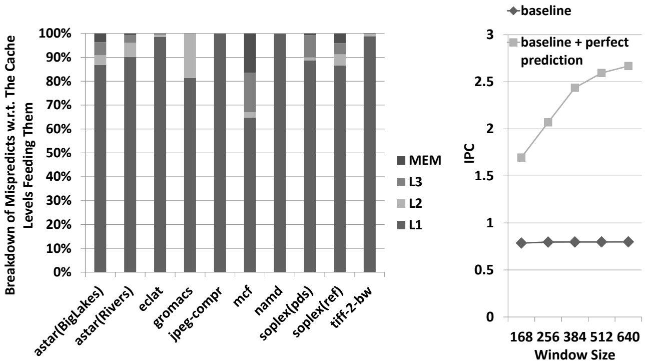
(a) IPC

(b) Energy of perfect branch prediction relative to real branch prediction

Figure 1: Impact of perfect branch prediction.

Some of these applications also suffer frequent last-level cache misses. Complexity-effective large-window processors can tolerate long-latency misses and exploit memory-level parallelism with small cycle-critical structures [31, 23]. Their ability to form an effective large window is degraded, however, when a mispredicted branch depends on one of the misses [31]. Figure 2a shows the breakdown of mispredicted branches that depend on data at various levels in the memory hierarchy: L1, L2, L3 and main memory. Figure 2b shows how the IPC of ASTAR (an application with high misprediction rate and significant fraction of mispredictions fed by L3 or main memory) scales with window size. Without perfect branch prediction, IPC does not scale with window size: miss-dependent branch mispredictions prevent a large window from performing its function of latency tolerance. Conversely, eradicating mispredictions acts as a catalyst for latency tolerance. IPC scales with window size in this case.

We first explore the current control-flow landscape by characterizing mispredictions in four benchmark suites using a state-of-art predictor. In particular, we classify the control-dependent regions guarded by hard-to-predict branches. About a third of mispredictions-per-1K-instructions (MPKI) come from branches with small control-



(a) Breakdown of mispredictions with respect to the furthest memory hierarchy level feeding them

(b) ASTAR(Rivers) IPC under different window sizes

Figure 2: Effect of branch mispredictions on memory latency tolerance.

dependent regions, e.g., hammocks. If-conversion using conditional moves, a commonly available predication primitive in commercial instruction-set architectures (ISA), is generally profitable for this class [2]. For completeness, we analyze why the gcc compiler did not if-convert such branches and manually do so at the source level in order to focus on other classes. We discover that another third of MPKI comes from what we call *separable* branches. A separable branch has two qualities:

1. The branch has a large control-dependent region, not suitable for if-conversion.
2. The branch does not depend on its own control-dependent instructions via a loop-carried data dependence (*totally separable*), or has only a short loop-carried dependence with its control-dependent instructions (*partially separable*).

For a totally separable branch, the branch's predicate computation is totally independent of the branch and its control-dependent region. This suggests "vectorizing" the control-flow: first generate a vector of predicates and then use this vector to drive fetching or skipping successive dynamic instances of the control-dependent region. This is the essence of our proposed technique, control-flow decoupling (CFD), for eradicating mispredictions of separable branches. The loop containing the branch is separated into two loops: a first loop contains only the instructions needed to compute the branch's predicate (generate branch outcomes) and a second loop contains the branch and its control-dependent instructions (consume branch outcomes). The first loop communicates branch outcomes to the second loop through an architectural queue, specified in the ISA and managed by push and pop instructions. At the microarchitecture level, the queue resides in the fetch unit to facilitate timely, non-speculative branching.

Partially separable branches can also be handled. In this case, the branch's predicate computation depends on some of its control-dependent instructions. This means a copy of the branch and the specific control-dependent instructions must be included in the first loop. Fortunately, this copy of the branch can be profitably removed by if-conversion due to few control-dependent instructions.

Either the programmer or compiler can transform a loop for CFD, and we evaluate both. On a microarchitecture configured similar to Intel's Sandy Bridge core [35], CFD increases performance by up to 43%, and reduces energy consumption by up to 41%. For hard-to-predict branches that traverse large data structures that suffer many cache misses, CFD acts as the necessary catalyst for future large-window architectures to tolerate these misses.

The paper is organized as follows. In Section 2, we discuss our methodology and classification of control-flow in a wide range of applications. In Section 3, we present the ISA, hardware and software aspects of CFD. In Section 4, we describe our implementation of CFD in the gcc compiler. In Section 5, we describe our evaluation framework and baseline selection process. In Section 6, we present an evaluation of the proposed techniques. In Section 7, we discuss prior related work. We conclude the paper in Section 8.

2. Methodology and Control-Flow Classification

The goal of the control-flow classification is first and foremost discovery: to gain insight into the nature of difficult branches' control-dependent regions, as this factor influences the solutions that will be needed, both old and new. Accordingly we cast a wide net to expose as many control-flow idioms as possible: (1) we use four benchmark suites comprised of over 80 applications, and (2) for the purposes of this comprehensive branch study, each application is run to completion leveraging a PIN-based branch profiling tool.

2.1. Methodology

We use four benchmark suites: *SPEC2006* [32] (engineering, scientific, and other workstation type benchmarks), *NU-MineBench-3.0* [24] (data mining), *BioBench* [1] (bioinformatics), and *cBench-1.1* [10] (embedded). All benchmarks[1] are compiled for x86 using gcc with optimization level *-O3* and run to completion using PIN [20]. We wrote a pintool that instantiates a state-of-art branch predictor (winner of CBP3, the third Championship Branch Prediction: 64KB ISL-TAGE [28]) that is used to collect detailed information for every static branch.

Different benchmarks have different dynamic instruction counts. In the misprediction contribution pie charts that follow, we weigh each benchmark equally by using its MPKI instead of its total number of mispredictions. Effectively we consider the average one-thousand-instruction interval of each benchmark.

Figure 3a shows the relative misprediction contributions of the four benchmark suites. Every benchmark of every suite is included[2], and, as just mentioned, each benchmark is allocated a slice proportional to its MPKI. We further refine the breakdown of each benchmark suite slice into *targeted* versus *excluded*, shown in Figure 3b. The excluded slice contains (1) benchmarks with misprediction rates less than 2%, and (2) benchmarks that we could not run in our detailed timing simulator introduced later (due to gcc Alpha cross-compiler problems). The targeted slice contains the remaining benchmarks. Table 1 lists the targeted benchmarks along with their MPKIs.

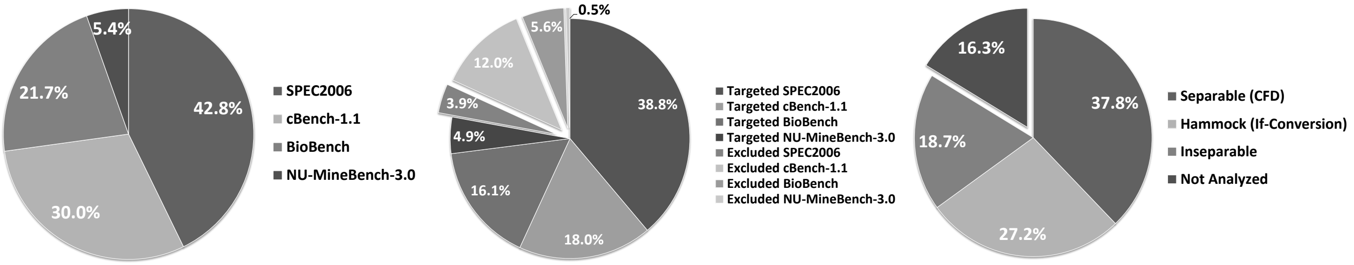
This paper focuses on the targeted slices which, according to Figure 3b, contribute almost 78% of cumulative MPKI in the four benchmark suites.

2.2. Control-Flow Classification

We inspected branches in the targeted benchmarks, and categorized them into the following four classes:

[1] For benchmarks with multiple ref inputs, we profiled then classified all inputs into groups based on the control-flow patterns exposed. One input is selected from each group in order to cover all observed patterns. For example, for *bzip* we select the ref inputs *input.source* and *chicken*.

[2] A benchmark that is present in multiple suites is included once. For example, *hmmer* appears in *BioBench* and *SPEC2006*. In both benchmark suites, the same hard-to-predict branches are exposed, thus, only one instance of *hmmer* is included.



(a) Mispredictions per benchmark suite

(b) Targeted vs. excluded mispredictions

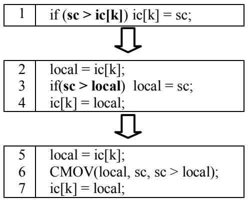
(c) Breakdown of targeted mispredictions

Figure 3: Breakdown of branch mispredictions.

Benchmark Suite	Application	MPKI	Benchmark Suite	Application	MPKI
SPEC2006	astar (BigLakes)	10.11	cBench	gsm	2.10
	astar (Rivers)	25.98		jpeg-compr	8.17
	bzip2 (chicken)	4.08		jpeg-decompr	2.41
	bzip2 (input.source)	8.16		quick-sort	4.64
	gobmk	7.17		tiff-2-bw	5.42
	gromacs	1.13		tiff-median	3.60
	hmmer	11.72			
	mcf	9.06	BioBench	clustalw	4.25
	namd	1.17		fasta	16.64
	sjeng	5.15			
	soplex (pds)	6.14	MineBench	eclat	10.19
	soplex (ref)	2.25			

Table 1: Targeted applications.

1. *Hammock*: Branches with small, simple control-dependent regions. Such branches will be if-converted. From what we can tell, the gcc compiler did not if-convert these branches because they guard stores. An example from *hmmer* is shown in Figure 4, line 1. To encourage if-conversion, the code can be adjusted (manually or using compiler) to unconditionally perform the store, if legal (i.e., if address is legal regardless of branch outcome). The control-dependent store to *ic[k]* (line 1) is moved outside the hammock (line 4) and the value being stored is a new local variable, *local*. Depending on the branch, *local* contains either the original value of *ic[k]* (line 2) or *sc* (line 3). Thus, the store to *ic[k]*, after the hammock, is effectively conditional – *ic[k]*'s value may or may not change – even though it is performed unconditionally. The new if-statement (line 3) is then if-converted by the compiler using a conditional move (line 6): conditionally move *sc* into *local* based on the condition *sc* > *local*. This transformation increases the number of retired stores, but the extra stores are silent. Obtaining the original value at the memory location requires a load, but we observed that most cases are like the *hmmer* example, in which the load already exists because the branch's test depends on a reference to *ic[k]* (line 1).
2. *Separable*: Branches with large, complex control-dependent regions, where the branch's backward slice (predicate computation) is either *totally separable* or *partially separable* from the branch and its control-dependent instructions. The backward slice is *totally separable* if it does not contain any of the branch's control-dependent instructions. Total separability allows all iterations of the backward slice to be hoisted outside the loop containing the branch, conceptually vectorizing the predicate computation, which is what CFD does via its first and second loops. The backward slice is *partially separable* if it contains very few of the branch's control-dependent instructions. In this case, the backward slice also contains the branch itself, since the branch guards the few control-dependent instructions in the slice. All iterations of the backward slice can still be hoisted but it contains a copy of the branch, therefore, the backward slice is if-converted. CFD will be applied to totally and partially separable branches.



```
1  if (sc > ic[k]) ic[k] = sc;

2  local = ic[k];
3  if(sc > local)  local = sc;
4  ic[k] = local;

5  local = ic[k];
6  CMOV(local, sc, sc > local);
7  ic[k] = local;
```

Figure 4: Hammock (from *HMMER*).

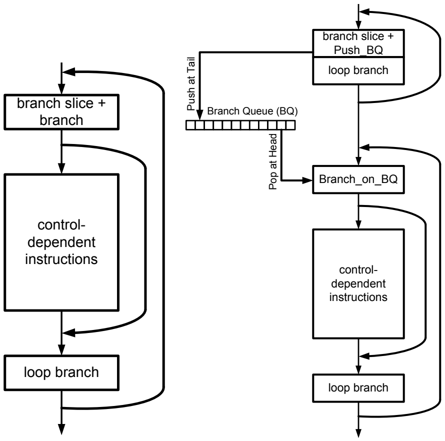
3. *Inseparable*: Branches with large, complex control-dependent regions, where the branch's backward slice contains too many of the branch's control-dependent instructions. An *inseparable* branch differs from a *partially separable* branch, in that it is not profitable to if-convert its backward slice. This type of branch is very serial in nature: the branch is frequently mispredicted and it depends on many of the instructions that it guards. This class of branch cannot be handled by if-conversion or CFD, and will require a new solution which is outside the scope of this paper.
4. *Not Analyzed*: Branches we did not analyze, i.e., branches with small contributions to total mispredictions.

Figure 3c breaks down the *targeted* mispredictions of Figure 3b into these four classes. 37.8% of the targeted mispredictions can be handled using CFD. 27.2% of the targeted mispredictions can be handled using if-conversion. That CFD covers the largest percentage of MPKI after applying a sophisticated branch predictor, provides a compelling case for CFD software, architecture, and microarchitecture support. Its applicability is on par with if-conversion, a commercially mainstream technique that also combines software, architecture, and microarchitecture. In addition to comparable MPKI coverage, CFD and if-conversion apply to comparable numbers of benchmarks and static branches (see Table 5 in Section 6).

3. Control-Flow Decoupling

Figure 5a shows a high-level view of a totally separable branch within a loop. *Branch slice* computes the branch's predicate. Depending on the predicate, the branch is taken or not-taken, causing its control-dependent instructions to be skipped or executed, respectively. In this example, none of the branch's control-dependent instructions are in its backward slice, i.e., there isn't a loop-carried data dependency between any of the control-dependent instructions and the branch. A partially separable branch would look similar, except a small number of its control-dependent instructions would be in the branch slice; this would appear as a backward dataflow edge from these instructions to the branch slice.

Figure 5b shows the loop transformed for CFD. The loop is separated into two loops, each with the same trip-count as the original. The first loop has just the branch slice. It pushes predicates onto an



(a) Original loop (b) CFD loops

Figure 5: High-level view of the CFD transformation.

architectural *branch queue* (BQ) using a new instruction, Push_BQ. The second loop has the control-dependent instructions. They are guarded by a new instruction, Branch_on_BQ. This instruction pops predicates from BQ and the predicates control whether or not the branch is taken.

Hoisting all iterations of the branch slice creates sufficient *fetch separation* between a dynamic instance of the branch and its producer instruction, ensuring that the producer executes before the branch is fetched. If successive iterations are a, b, c, ..., instead of fetching *slice-a, branch-a, slice-b, branch-b, slice-c, branch-c, ...*, the processor fetches *slice-a, slice-b, slice-c, ... branch-a, branch-b, branch-c, ...* Additionally, to actually exploit the now timely predicates, they must be communicated to the branch in the fetch stage of the pipeline so that the branch can be resolved at that time. Communicating through the existing source registers would not resolve the branch in the fetch stage. This is why we architect the BQ predicate communication medium and why, microarchitecturally, it resides in the fetch unit.

While this paper assumes an OOO processor for evaluation purposes, please note that in-order and OOO processors both suffer branch penalties due to the fetch-to-execute delay of branches. We want to resolve branches in the fetch stage (so fetching is not disrupted) but they resolve in the execute stage, unless correctly predicted. Thus, the problem with branches stems from pipelining in general. OOO execution merely increases the pipeline's speculation depth (via buffering in the scheduler) so that, far from being a solution to the branch problem, OOO execution actually makes the branch problem more acute.

For a partially separable branch, the first loop would not only have (1) the branch slice and Push_BQ instruction, but also (2) the branch and just those control-dependent instructions that feed back to the branch slice. The branch is then removed by if-conversion, using conditional moves to predicate the control-dependent instructions. CFD is still profitable in this case because the subsetted control-dependent region is small and simple (otherwise the branch would be classed as inseparable).

CFD is a software-hardware collaboration. The following subsections discuss ISA, software, and hardware.

3.1. ISA Support and Benchmark Example

ISA support includes an architectural specification of the BQ and two instructions, Push_BQ and Branch_on_BQ. The architectural specification of the BQ is as follows:

1. The BQ has a specific size. BQ size has implications for software. These are discussed in the next subsection.
2. Each BQ entry contains a single flag indicating taken/not-taken (the predicate). Other microarchitectural state may be included in each entry of the BQ's physical counterpart, but this state is transparent to software and not specified in the ISA.
3. A length register indicates the BQ occupancy. Architecting only a length register has the advantage of leaving low-level management concerns to the microarchitect. For example, the BQ could be implemented as a circular or shifting buffer. Thus, at the ISA level, the BQ head and tail are conceptual and are not specified as architectural registers: their physical counterparts are implementation-dependent.
4. The ISA provides mechanisms to save and restore the BQ state (queue contents and length register) to memory. This is required for context-switches. We recommend the approach used in some commercial ISAs, which is to include the BQ among the special-purpose registers and leverage move-from and move-to special-purpose-register instructions to transfer the BQ state to and from general-purpose registers (which can be saved and restored via stores and loads, respectively). If this is not possible, then dedicated Save_BQ and Restore_BQ instructions could be used.

The Push_BQ instruction has a single source register specifier to reference a general-purpose register. If the register contains zero (non-zero), Push_BQ pushes a 0 (1). Branch_on_BQ is a new conditional branch instruction. Branch_on_BQ specifies its taken-target like other conditional branches, via a PC-relative offset. It does not have any explicit source register specifiers, however. Instead, it pops its predicate from the BQ and branches or doesn't branch, accordingly.

The ISA specifies key ordering rules for pushes and pops, that software must abide by. First, a push must precede its corresponding pop. Second, N consecutive pushes must be followed by exactly N consecutive pops in the same order as their corresponding pushes. Third, N cannot exceed the BQ size.

Figure 6 shows a real example from the benchmark *SOPLEX*. Referring to the original code: The loop compares each element of array *test[]* to variable *theeps*. The hard-to-predict branch is at line 3 and its control-dependent instructions are at lines 4-9. Neither the array nor the variable is updated inside the control-dependent region, thus, this is a totally separable branch. This branch contributes 31% of the benchmark's mispredictions (for *ref* input).

Decoupling the loop is fairly straightforward. The first loop computes predicates (lines 2-3) and pushes them onto the BQ (line 4). The second loop pops predicates from the BQ and conditionally executes the control-dependent instructions, accordingly (line 7).

An ISA enhancement must be carefully specified, so that its future obsolescence does not impede microarchitects of future generation processors. Accordingly, CFD is architected as an optional and scalable co-processor extension:

1. Optional: Inspired by configurability of co-processors in the MIPS ISA – which specifies optional co-processors 1 (floating-point unit) and higher (accelerators) – BQ state and instructions can be encapsulated as an optional co-processor ISA extension. Thus, future implementations are not bound by the new BQ co-processor ISA. Codes compiled for CFD must be recompiled for processors that do not implement the BQ co-processor ISA, but this is no different than the precedent set by MIPS' flexible co-processor specification.
2. Scalable: The BQ co-processor ISA can specify a BQ size of

Original Loop

```
for ( … ) {
  x = test[i];
  if (x < -theeps) {                          // hard-to-predict branch
    x *= x / penalty_ptr[i];
    x *= p[i];
    if (x > best) {                           // predictable branch
      best = x;
      selId = thesolver->id(i);
    }
  }
}
```

Decoupled Loops

First Loop

```
for ( … ) {
 x = test[i];
 pred = (x < -theeps);                        // the predicate is computed
 Push_BQ(pred);                               // then pushed onto the BQ
}
```

Second Loop

```
for ( … ) {
 Branch_on_BQ{                                // pop the predicate
    x = test[i];
    x *= x / penalty_ptr[i];
    x *= p[i];
    if (x > best) {
      best = x;
      selId = thesolver->id(i);
    }
  }
}
```

Figure 6: SOPLEX' source code.

N: a machine-dependent parameter, thus allowing scalability to different processor window sizes.

3.2. Software Side

For efficiency, the trip-counts of the first and second loops should not exceed the BQ size. This is a matter of performance, not correctness, because software can choose to spill/fill the BQ to/from memory. In practice, this is an important issue because many of the CFD-class loops iterate thousands of times whereas we specify a BQ size of 128 in this paper.

We explored multiple solutions but the most straightforward one is loop strip mining. The original loop is converted to a doubly-nested loop. The inner loop is similar to the original loop but its trip-count is bounded by the BQ size. The outer loop iterates a sufficient number of times to emulate the original loop's trip-count. Then, CFD is applied to the inner loop.

Decoupling the loop can be done either manually by the programmer or automatically by the compiler. In this paper, CFD was initially applied manually which was a fairly easy task. In Section 4, we describe automating CFD in the gcc compiler and in Section 6 we evaluate how well it compares to the manual implementation.

3.3. Hardware Side

This subsection describes microarchitecture support for CFD. The BQ naturally resides in the instruction fetch unit. In our design, the BQ is implemented as a circular buffer. In addition to the software-visible predicate bit, each BQ entry has the following microarchitectural state: pushed bit, popped bit, and checkpoint id. For a correctly written program, a Push_BQ (push) instruction is guaranteed to be fetched before its corresponding Branch_on_BQ (pop) instruction. Because of pipelining, however, the push might not execute before the pop is fetched, referred to as a late push. The pushed bit and popped bit enable synchronizing the push and pop. We explain BQ operation separately for the two possible scenarios: early push and late push.

3.3.1. Early Push. The early push scenario is depicted in Figure 7, left-hand side.

When the push instruction is fetched, it is allocated the entry at the BQ tail. It initializes its entry by clearing the pushed and popped bits. The push instruction keeps its BQ index with it as it flows down the pipeline [3]. When the push finally executes, it checks the popped bit in its BQ entry. It sees that the popped bit is still unset. This means the scenario is early push, i.e., the push executed before its pop counterpart was fetched. Accordingly, the push writes the predicate into its BQ entry and sets the pushed bit to signal this fact.

Later, the pop instruction is fetched. It is allocated the entry at the BQ head, which by the ISA ordering rules must be the same entry as its push counterpart. It checks the pushed bit. It sees that the pushed bit is set, therefore, it knows to use the predicate that was pushed earlier. The pop executes right away, either branching or not branching according to the predicate.

3.3.2. Late Push. The late push scenario is depicted in Figure 7, right-hand side.

In this scenario, the pop is fetched before the push executes. As before, when the pop is fetched, it checks the pushed bit to see if the push executed. In this case the pushed bit is still unset so the pop knows that a predicate is not available. There are two options: (1) stall the fetch unit until the push executes, or (2) predict the predicate using the branch predictor. Our design implements option 2 which we call a *speculative pop*. When the speculative pop reaches the rename stage, a checkpoint is taken. (This is on top of the baseline core's branch checkpointing policy, which we thoroughly explore in Section 5.) Unlike conventional branches, the speculative pop cannot confirm its prediction – this task rests with the late push instruction. Therefore, the speculative pop writes its predicted predicate and checkpoint id into its BQ entry, and signals this fact by setting the popped bit. This information will be referenced by the late push to confirm/disconfirm the prediction and initiate recovery if needed.

When the push finally executes, it notices that the popped bit is set in its BQ entry, signifying a late push. The push compares its predicate with the predicted one in the BQ entry. If they don't match, the push initiates recovery actions using the checkpoint id that was placed there by the speculative pop. Finally, the push writes the predicate into its BQ entry and sets the pushed bit.

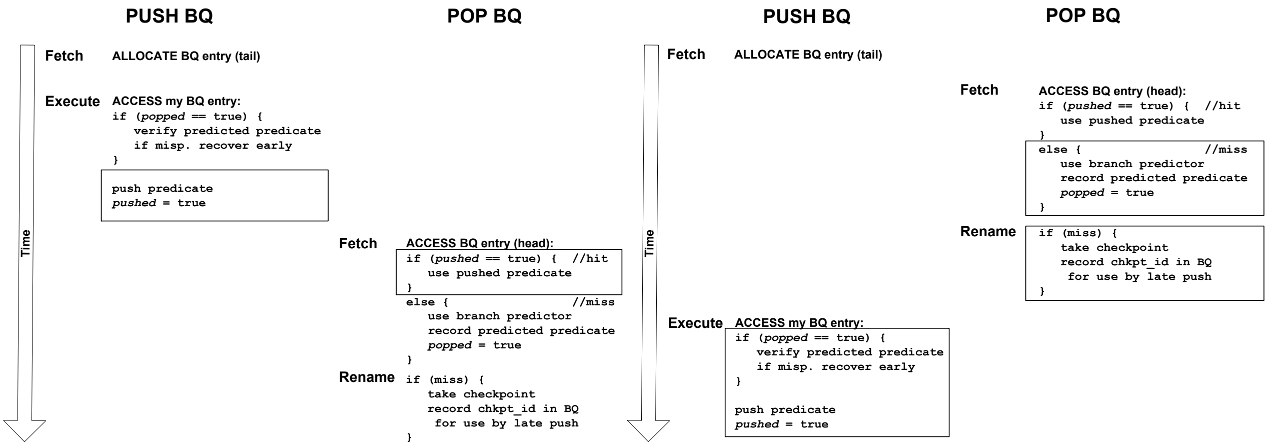
Empirically, late pushes are very rare in our CFD-modified benchmarks, less than 0.1% of pops (one per thousand). When fully utilized by software, a 128-entry BQ separates a push and its corresponding pop by 127 intervening pushes. This typically corresponds to a push/pop separation of several hundreds of instructions, providing ample time for a push to execute before its pop counterpart is fetched.

3.3.3. BQ Length. The BQ length (occupancy) is the sum of two components:

1. net_push_ctr: This is the net difference between the number of pushes and pops retired from the core up to this point in the program's execution. The ISA push/pop ordering rules guarantee this count will always be greater than or equal to zero and less than or equal to BQ size. This counter is incremented when a push retires and decremented when a pop retires.
2. pending_push_ctr: This is the number of pushes in-flight in the window, i.e., the number of fetched but not yet retired pushes. It is incremented when a push is fetched, decremented when a push is retired (because it now counts against net_push_ctr), and possibly adjusted when a mispredicted branch resolves (see next section).

BQ length must be tracked in order to detect the BQ stall condition. In particular, if BQ length is equal to BQ size and the fetch unit

[3] Having the BQ index in the push instruction's payload enables it to reference its BQ entry later, when it executes OOO. This is a standard technique for managing microarchitecture FIFOs such as the reorder buffer and load and store queues.



```
ALLOCATE BQ entry (tail)

ACCESS my BQ entry:
if (popped == true) {
   verify predicted predicate
   if misp. recover early
}

push predicate
pushed = true
```

```
ACCESS BQ entry (head):
if (pushed == true) {  //hit
   use pushed predicate
}
else {                 //miss
   use branch predictor
   record predicted predicate
   popped = true
}

if (miss) {
   take checkpoint
   record chkpt_id in BQ
    for use by late push
}
```

Figure 7: BQ operation. Two scenarios are shown: early push (left, common) and late push (right, uncommon).

fetches a push instruction, the fetch unit must stall. Note that the stall condition is guaranteed to pass for a bug-free program. The ISA push/pop ordering rules guarantee that there are *BQ size* in-flight pop instructions prior to the stalled push. The first one of these pops to retire will unstall the stalled push.

3.3.4. BQ Recovery. The core may need to roll back to a branch checkpoint, in the case of a mispredicted branch, or the committed state, in the case of an exception. In either case, the BQ itself needs to be repaired.

1. Preparing for misprediction recovery: Each branch checkpoint is augmented with state needed to restore the BQ to that point in the program execution. Namely, in addition to the usual checkpointed state (Rename Map Table, etc.), each checkpoint also takes a snapshot of the BQ head and tail pointers. This is a modest amount of state compared to other checkpointed state.
2. Preparing for exception recovery: Exception recovery requires maintaining committed versions of the BQ head and tail pointers, called arch_head and arch_tail. Arch_head and arch_tail are incremented when pops and pushes retire, respectively.

When there is a roll-back, the BQ head and tail pointers are restored from the referenced checkpoint (on a misprediction) or their committed versions (on an exception), and all popped bits between the restored head and tail are cleared. Moreover, pending_push_ctr (the second component of BQ length) is reduced by the number of entries between the tail pointers before and after recovery (this corresponds to the number of squashed push instructions).

3.3.5. Branch Target Buffer. Like all other branch types, Branch_on_BQ is cached in the fetch unit's Branch Target Buffer (BTB) so that there is no penalty for a taken Branch_on_BQ as long as the BTB hits. The BTB's role is to detect branches and provide their taken-targets, in the same cycle that they are being fetched from the instruction cache. This information is combined with the taken/not-taken prediction (normal conditional branch) or the popped predicate (Branch_on_BQ) to select either the sequential or taken target. As with other branches, a BTB miss for a taken Branch_on_BQ results in a 1-cycle misfetch penalty (detected in next cycle).

Predicates for potential Branch_on_BQ instructions in the current fetch bundle are obtained from the BQ in parallel with the BTB access, because these predicates are always at consecutive entries starting at the BQ head.

3.4. Optimization

This section describes an optimization on top of CFD, that can reduce CFD instruction overheads in some cases. We observed that values used to compute the predicate in the first loop are used again, thus recomputed, inside the control-dependent region in the second loop. A simple way to avoid duplication is to communicate values from the first loop to the second loop using an architectural value queue (VQ) and VQ push/pop instructions. We call this optimization CFD+.

An interesting trick to leverage existing instruction issue and register communication machinery in a superscalar core, is to map the architectural value queue onto the physical register file. This is facilitated by the *VQ renamer* in the rename stage. The VQ renamer is a circular buffer with head and tail pointers. Its entries contain physical register mappings instead of values. The mappings indicate where the values are in the physical register file. A VQ push is allocated a destination physical register from the freelist. Its mapping is pushed at the tail of the VQ renamer. A VQ pop references the head of the VQ renamer to obtain its source physical register mapping. The queue semantics ensure the pop links to its corresponding push through its mapping. In this way, after renaming, VQ pushes and pops synchronize in the issue queue and communicate values in the execution lanes the same way as other producer-consumer pairs. The physical registers allocated to push instructions are freed when the pops that reference them retire.

4. CFD Compiler Implementation

We have implemented a compiler pass to perform the CFD code transformation automatically. The pass needs a list of hard-to-predict branches derived from profiling or the programmer as input, and it transforms the inner-loop containing the branch into CFD form. We will refer to the decoupled first and second loops created by the compiler pass as the *Producer* and *Consumer* loops, respectively. Algorithm 1 shows the overall CFD compiler implementation. We start with the CFD function which takes as input a loop and the hard-to-predict predicates that are contained within the loop.

The first steps of the algorithm are inspired by the Decoupled Software Pipelining (DSWP) algorithm presented by Ottoni *et al.* [25]. In particular, we borrow their strategy of first constructing a full Program Dependence Graph (PDG) and then consolidating the strongly connected components (SCCs) into single nodes in the graph to create a directed acyclic graph (Lines 2-3). If the hard-to-predict branch forms the root of a control-dependent region which can be

Algorithm 1 Overall CFD algorithm.

```
1: function CFD(Loop l, Predicate p)
2:     pdg ← BuildPDG(l)
3:     dag ← ConsolidateSCCs(pdg)
4:     MarkPredicateSlices(dag,p)
5:     AssignStmtsToLoops(dag)
6:     if non-empty CFD region found in dag then
7:         producer ← l
8:         consumer ← CloneLoop(l)
9:         ConnectLoops(producer,consumer)
10:        for all control flow decoupled branches, b do
11:            Insert Push_BQ(Predicate(b)) in producer just before b
12:            Replace b in consumer with Branch_on_BQ
13:        end for
14:        for all r, def'ed in producer and used in consumer do
15:            Insert push in producer at definition of r
16:            Replace definition of r in consumer with "r=Pop_VQ()"
17:        end for
18:        Remove code from producer assigned only to consumer
19:        Remove code from consumer assigned only to producer
20:        Final Dead and Redundant Code Elimination
21:    end if
22: end function
```

isolated into one or more SCCs, then the branch is separable. We use the consolidated graph to assign nodes to the Producer and Consumer loops. Otherwise, if the control-dependent code is part of the same SCC as the loop's exit condition, then no decoupling may be possible (and our algorithm gives up).

In Line 4, we call the MarkPredicateSlices subroutine which carries out the following operations. For each predicate in the loop, it finds its corresponding node in the *dag*. All nodes in its forward slice (all immediate successors and those reached through a depth-first search (DFS)) are marked as belonging to the Consumer. All nodes in its backward slice and itself (all immediate predecessors and those reached through a reverse DFS) are marked as belonging to the Producer.

At this point, some nodes in *dag* have been scheduled among the Producer and Consumer loops, but many nodes may remain unscheduled. For example, any node that is both control independent and data independent from marked nodes will need to be assigned a loop. AssignStmtsToLoops, on line 5, completes the task of scheduling.

AssignStmtsToLoops. This function simultaneously solves several problems. First, it must create a correct schedule. A statement must be placed in the Producer if any dependent instruction has already been placed in the Producer. Similarly, a statement must be placed in the Consumer if any statement it depends upon has already been placed in the Consumer. These rules must always be enforced. Fortunately, some flexibility does exist that can be leveraged for optimization. For example, if a statement produces no side-effects, then we can optionally schedule it in both loops. This flexibility allows to choose between replicating work and communicating values depending on which is more efficient. We use a simple heuristic to solve both problems at once as shown in Algorithm 2.

In lines 2-7, each node is initially marked as NoReplicate to mean that it must be scheduled in only one loop. Next, we figure out if the node has any side-effects (stores or function calls) which would prevent replication, and if it does not, it is marked MaybeReplicate to mean that we can possibly schedule it in both loops.

The second for-all loop visits all nodes in topological order, which means we must visit a node's predecessors in an earlier iteration. This makes it easy to reason about predecessors since they have already been processed.

In lines 9-15, we assign a node to a loop if it was not assigned one in MarkPredicateSlices. Note, we prefer to place a node in the Producer unless forced to place it in the Consumer. Once we know where the node will be scheduled, we need to determine if it is better to communicate or replicate any values it produces for the Consumer loop. Lines 16-23 form this judgement. First, we check to make sure that the node is in the Producer and that it is marked MaybeReplicate. To determine if we should replicate, we compare the estimated runtime cost (EstCost) of the node against a minimal threshold that determines when communication will be cheaper. If the node is expensive to execute, we mark the node as NoReplicate which means that any register it defines must be communicated to the Consumer loop. Otherwise, we mark the node as Replicate and it will be computed in both loops. For all of our results, we use CommThreshold=2.

Algorithm 2 Assign all statements to the Producer and/or Consumer loops.

```
1: function AssignStmtsToLoops(PDG dag)
2:     for all $n \in dag$ do
3:         Mark n as NoReplicate
4:         if n has no side-effects then
5:             Mark n as MaybeReplicate
6:         end if
7:     end for
8:     for all $n \in dag$, in topological order do
9:         if n has not been placed in a loop then
10:            if any predecessor of n is in the Consumer then
11:                place n in the Consumer
12:            else
13:                place n in the Producer
14:            end if
15:        end if
16:        if n placed in Producer and n marked MaybeReplicate then
17:            if n communicates to Consumer
18:                and EstCost(n) > CommThreshold then
19:                n marked NoReplicate (values will be communicated)
20:            else
21:                n marked Replicate
22:            end if
23:        end if
24:    end for
25: end function
```

Final Code Generation. If a non-empty CFD region is found (line 6 of Algorithm 1), we finalize the loops and generate the code. This process is shown in lines 7-20. First, we clone the loop (line 8) and use the original as the Producer and the clone as the Consumer. Next, we connect the loops so that the program will first execute the Producer and then the Consumer. This entails redirecting the exits of the Producer to the pre-header of the Consumer. The Consumer's exits, since they are a clone of the original loop, remain unchanged. Also, our implementation works on an SSA graph, so we also fix the phi-nodes at the pre-header of the Consumer loop and exits of the Consumer loop. Also while connecting the loops, we perform the loop strip mining transformation described in Section 3.2. This is easily accomplished by inserting a new outer loop to surround both the Producer and Consumer and by forcing a break from each loop every *BQ size* iterations. Early breaks/returns are handled by keeping a loop count in the Producer and passing that count to the Consumer for it to use as its trip count.

Next, we insert the necessary predicate and value communication. In lines 10-13, we visit all predicates that are computed in the Producer loop and communicated to the Consumer loop. We insert a Push_BQ in the Producer and place a Branch_on_BQ in the Consumer in place of the original branch. This loop will always handle the original hard-to-predict predicates but additional predicates may also be included if the partitioning algorithm places a predicate in the Producer that has control-dependent instructions in the Consumer. Ideally, the partitioning algorithm should limit the frequency of this case.

In lines 14-17, we insert value communication between the pro-

	AMD Bobcat	ARM Cortex A15	IBM Power6	INTEL Pentium 4
Fetch-to-Execute	13	15	13	20

Table 2: Minimum fetch-to-execute latency in cycles.

ducer and consumer loops. Any register that is defined in the Producer and used in the Consumer must be communicated. The push is placed at the definition in the Producer and the pop is placed at the same point (the cloned register definition) in the Consumer.

Finally, the Producer and Consumer code is cleaned up. All instructions assigned the Consumer partition are removed from the Producer loop and vice versa for the Consumer loop. Then, a final dead and redundant code elimination pass eliminates other inefficiencies, like empty basic blocks and useless control-flow paths.

5. Evaluation Environment

The microarchitecture presented in Section 3 is faithfully modeled in a detailed execution-driven, execute-at-execute, cycle-level simulator. The simulator runs Alpha ISA binaries. Recall, in Section 2, we used x86 binaries to locate hard-to-predict (easy-to-predict) branches, owing to our use of PIN. Our collected data confirms that hard-to-predict (easy-to-predict) branches in x86 binaries are hard-to-predict (easy-to-predict) in Alpha binaries. The predictability is influenced far more by program structure than the ISA that it gets mapped to.

Section 2 described the four benchmark suites used. All benchmarks are compiled to the Alpha ISA using gcc with -O3 level optimization. (We built gcc from scratch using the trunk SVN repository in the gcc-4 development line.) When applied, if-conversion and CFD modify the benchmark source. The modified benchmarks are verified by compiling natively to the x86 host, running them to completion, and verifying outputs (software queues are used to emulate the CFD queues).

Energy is measured using McPAT [19], which we augmented with energy accounting for the BQ (CFD, CFD+) and VQ (CFD+). Per-access energy for the BQ and VQ is obtained from CACTI tagless rams, and every read/write access is tracked during execution.

The parameters of our baseline core are configured as close as possible to those of Intel's Sandy Bridge core [35]. The baseline core uses the state-of-art ISL-TAGE predictor [28]. Additionally, in an effort to find the best-performing baseline, we explored the design space of misprediction recovery policies, including checkpoint policies (in-order vs. OoO reclamation, with confidence estimator [13] versus without) and number of checkpoints (from 0 to 64). We confirmed that: (1) An aggressive policy (OoO reclamation, confidence-guided checkpointing) performs best. (2) The harmonic mean IPC, across all applications of all workloads, levels off at 8 checkpoints.

The fetch-to-execute pipeline depth is a critical parameter as it factors into the branch misprediction penalty. Table 2 shows the minimum fetch-to-execute latency (number of cycles) for modern processors from different vendors. The latency ranges from 13 to 20 cycles [6, 17, 18, 7]. We conservatively use 10 cycles for this parameter. We also perform a sensitivity study with this parameter in Section 6.1.1.

Table 3 shows the baseline core configuration. The checkpoint management policy and number of checkpoints remain unchanged throughout our evaluation, even for studies that scale other window resources.

6. Results and Analysis

To evaluate the impact of our work on the top contributors of branch mispredictions in the targeted applications, we identify the regions to be simulated as follows. Given the set of top mispredicting branches and the functions in which they reside, we fast-forward to the first occurrence of the first encountered function of interest, warm up for 10M retired instructions, and then simulate for a certain number of retired instructions. When simulating the unmodified binary for the baseline, we simulate 100M retired instructions. When simulating binaries modified for CFD or if-conversion, we simulate as many retired instructions as needed in order to perform the same amount of work as 100M retired instructions of the unmodified binary. Table 4 shows the fast-forward (skip) distances of the applications and the overheads incurred by the modified binaries. Overhead is the factor by which retired instruction count increases (e.g., 1.5 means 1.5 times) for the same simulated region. In all cases except SOPLEX(ref), the modified binaries are simulated for more than 100M retired instructions[4]. Speedup is calculated as: $cycles_{baseline}/cycles_{CFD}$, where $cycles_{baseline}$ is the number of cycles to simulate 100M instructions of the unmodified binary and $cycles_{CFD}$ is the number of cycles to simulate overhead_factor x 100M instructions of the CFD-modified binary which corresponds to the same simulated region.

Table 5 shows detailed information about the modified source code, most importantly: (1) the affected branches and (2) the fraction of time spent in the functions containing these branches, as found by gprof-monitored native execution[5].

Branch Prediction	**BP**: 64KB ISL-TAGE predictor - 16 tables: 1 bimodal, 15 partially-tagged. In addition to, IUM, SC, LP. - History lengths: {0, 3, 8, 12, 17, 33, 35, 67, 97, 138, 195, 330, 517, 1193, 1741, 1930} **BTB**: 4K entries, 4-way set-associative **RAS**: 64 entries
Memory Hierarchy	**Block size**: 64B **Victim caches**: each cache has a 16-entry FA victim cache **L1**: split, 64KB each, 4-way set-associative, 1-cycle access latency **L2**: unified, private for each core, 512KB, 8-way set-associative, 20-cycle access latency - **L2 stream prefetcher**: 4 streams, each of depth 16 **L3**: unified, shared among cores, 8MB, 16-way set-associative, 40-cycle access latency **Memory**: 200-cycle access latency
Fetch/Issue/Retire Width	4 instr./cycle
ROB/IQ/LDQ/STQ	168/54/64/36 (modeled after Sandy Bridge)
Fetch-to-Execute Latency	10-cycle
Physical RF	236
Checkpoints	8, OoO reclamation, confidence estimator (8K entries, 4-bit resetting counter, gshare index)
CFD	• **BQ**: 96B (128 6-bit entries) • **VQ renamer**: 128B (128 8-bit entries)

Table 3: Baseline core configuration.

Application	**Skip (B)**	**Overhead**		
		CFD	**CFD+**	**Compiler (CFD+)**
astar(BigLakes)	11.61	1.86	-	-
astar(Rivers)	0.53	1.81	-	-
eclat	7.10	1.28	1.12	1.14
gromacs	0.74	1.03	1.02	-
jpeg-compr	0.00	1.08	1.06	1.09
mcf	0.70	1.15	1.14	1.20
namd	2.17	1.01	-	-
soplex(pds)	9.94	1.02	1.02	1.04
soplex(ref)	49.25	0.90	-	1.41
tiff-2-bw	0.00	1.00	-	1.00
Application	**Skip (B)**	**If-Conversion**		
clustalw	0.04	1.0		
fasta	0.00	1.0		
gsm	0.00	1.03		
hmmer	0.02	1.0		
jpeg-decompr	0.00	1.0		
quick-sort	0.19	1.06		
sjeng	0.17	1.02		

Table 4: Application skip distances and overheads.

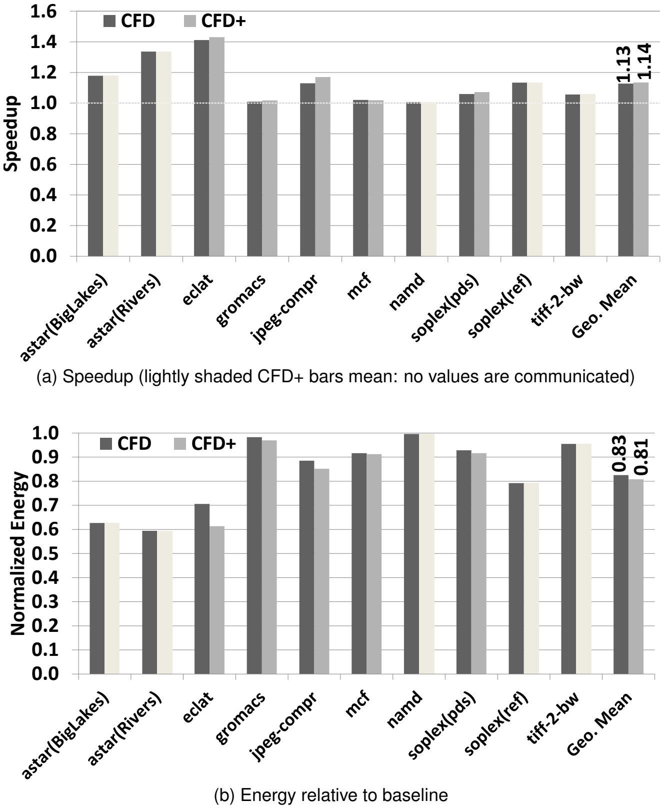
[4] SOPLEX(ref) is an exception. The original loop contains many variables whose live ranges overlap, increasing pressure on architectural registers and resulting in many stack spills/fills. CFD's two loops reduce register contention by virtue of some variables shifting exclusively to the first or second loop, eliminating most of the stack spills/fills, resulting in fewer retired instructions.

[5] The fraction of time spent in the function(s) of interest is found using *gprof* while running the x86 binaries (compiled using gcc with -O3) to completion 3 times on an idle, freshly rebooted Sandy Bridge Processor running in single-user mode.

Application	File name	Function	Time spent	Loop line	Branch line	Loop strip mining	Communicate values
astar	Way_.cpp	makebound2	20% (BigLakes) 47% (Rivers)	57	62-63, 79-80 96-97, 113-114 130-131, 147-148 164-165, 181-182	Y	N
eclat	eclat.cc	get_intersect	46%	205	207, 211	Y	Y
gromacs	ns.c	ns5_core	11%	1503	1507, 1508, 1510	N	Y
jpeg-compr	jcdctmgr.c	forward_DCT	83%	322	251	N	Y
	jcphuf.c	encode_mcu_AC_first		488	489		
		encode_mcu_AC_refine		662	663, 686		N
mcf	pbeampp.c	primal_bea_mpp	39%	165	171	Y	Y
namd	ComputeNonbondedBase.h	ComputeNonbondedUtil	5%	397	410	Y	N
soplex	spxsteeppr.cc	selectLeaveX	5% (pds) 17% (ref)	291	295	Y	Y
		selectEnterX		449	452		N
tiff-2-bw	tif_lzw.c	LZWDecode	100%	377	411	N	N

Application	File name	Function	Time spent	Branch Line
clustalw	pairalign.c	forward_pass	98%	384, 388, 391-393
		reverse_pass		436, 440, 443-444
		diff		526-527, 529-530 555-556, 558-559
fasta	dropnfa.c	FLOCAL_ALIGN	47%	1085-1086, 1096-1097 1099-1101, 1104
gsm	add.c	gsm_div	49%	228
	long_term.c	Calculation_of_the_LTP_parameters		152
hmmer	fast_algorithms.c	P7Viterbi	100%	135-137, 142, 147
jpeg-decompr	jdhuff.c	HUFF_EXTEND	50%	372
	jdphuff.c			207
quick-sort	qsort_large.c	compare	43%	26
sjeng	moves.c	make	10%	1305
	search.c	remove_one		515

Table 5: Details of modified code: CFD (left), If-Conversion (right).



(a) Speedup (lightly shaded CFD+ bars mean: no values are communicated)

(b) Energy relative to baseline

Figure 8: Performance and energy impact of CFD.

6.1. CFD

6.1.1. Manual CFD. We manually apply then evaluate: CFD and CFD+. Figure 8a shows that CFD increases performance by up to 41% and 13% on average, while CFD+ increases performance by up to 43% and 14% on average [6].

Figure 8b shows that CFD reduces energy consumption by up to 41% and 17% on average, while CFD+ reduces energy consumption by up to 41% and 19% on average.

Figure 9 shows speedup with CFD as the minimum fetch-to-execute latency is varied from five to twenty cycles. As expected, CFD gains increase as the pipeline depth increases. The baseline IPC worsens with increasing depth, whereas CFD's eradication of mispredicted branches makes IPC insensitive to pipeline depth. Thus, as is true with better branch prediction, CFD has the added benefit of exacting performance gains from frequency scaling (i.e., deeper pipelining).

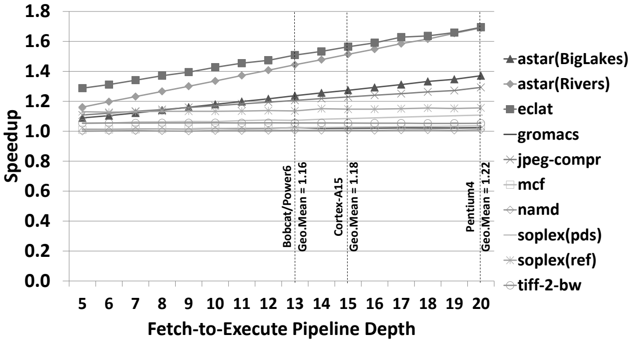


Figure 9: Varying the minimum fetch-to-execute latency.

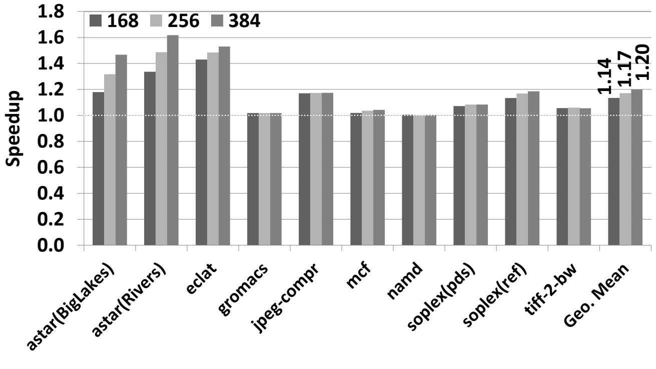


Figure 10: CFD speedups as we scale the processor structures.

To project the gains of CFD+ on future processor generations, we evaluate it under larger instruction windows. Figure 10 shows the projection of CFD gains on two additional configurations labeled in the graph with ROB size[7]. The average performance improvement increases to 20%.

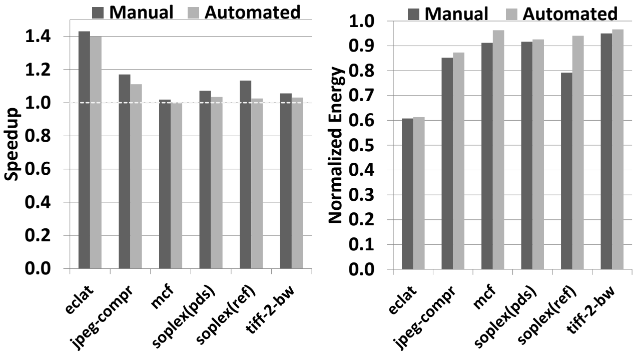
6.1.2. Automated CFD. We present results of our CFD compiler pass for six applications: ECLAT, JPEG, MCF, SOPLEX (pds and ref), and TIFF-2-BW. Figure 11 compares the performance improvements and energy savings of manual CFD+ vs. automated CFD+. The two approaches yield close results for five of the six applications. For SOPLEX(ref), the compiler was unable to register-promote a global variable accessed within the first loop, causing it to be repeatedly loaded within the loop, increasing the instruction overhead and decreasing speedup. The employed alias analysis cannot confirm that the global variable is not stored to by a store within the loop. Inspection of the whole benchmark gives us confidence that interprocedural alias analysis would be able to confirm safety of register-promoting the global variable, because its address is never taken.

As for the other four benchmarks:

1. NAMD, GROMACS: We simply have not yet attempted these

[6] The time spent in the functions of interest (shown in Table 5) along with the presented speedups, can be used in Amdahl's law to estimate the speedup of the whole benchmark. For example, ASTAR(Rivers) is sped up by 34% (s=1.34) in its CFD region which accounts for 47% of its original execution time (f=0.47); thus, we estimate 14% (1.14) speedup overall.

[7] [ROB,IQ,LDQ,STQ,PRF] are as follows for the two additional configurations: [**256**, 82, 96, 54, 324] and [**384**, 122, 216, 82, 452]. Other parameters match those of the baseline, shown in Table 3 in Section 5.



(a) Speedup (b) Energy relative to baseline

Figure 11: Comparison of manual and automated CFD.

benchmarks but do not anticipate difficulty. NAMD was deprioritized due to low MPKI and we recently added GROMACS to the mix.

2. ASTAR (Rivers and Biglakes): This benchmark has complexity that is not yet supported by our compiler pass: (1) It has a partially separable branch. Note that all other targeted benchmarks have only totally separable branches. (2) The control-dependent instruction in the branch's backward slice is a store, hence, if-converting the backward slice requires the transformation described in Section 2.2. (3) It has two nested, separable branches (one partially separable and one totally separable). We explore this complexity, in depth, in Section 6.1.3.

6.1.3. ASTAR Case Study. One of the most interesting cases we encountered in this work is ASTAR. Figure 12 shows a simplified version of ASTAR's original and decoupled loops.

ASTAR has a few challenging features that require special care when decoupling its loop. First, there are two nested hard-to-predict branches, with the inner predicate depending on a memory reference that is only safe if the outer predicate is true (lines 3 and 4 of original loop). Second, there is a short loop-carried dependency between the outer predicate and one of its control-dependent instructions (line 7 of original loop): this is a partially separable branch.

These challenges are naturally handled by CFD. The nested conditions are handled by decoupling the original loop into *three* loops. The first loop evaluates the outermost condition. The second loop, guarded by the outermost condition, evaluates the combined condition. The third loop guards the control-dependent instructions by the overall condition. The loop-carried dependency is handled by hoisting then if-converting the short loop-carried dependencies (shown in lines 12 and 13 of the second loop).

Due to the high percentage of branch mispredictions that are fed by the L3 cache and main memory, we expect a significant increase in performance gains when we apply CFD to ASTAR under large instruction windows. Figure 13 shows the effective IPC of the unmodified binaries (baseline) and the CFD binaries, as we scale the window size. Our expectations are confirmed for CFD.

6.2. If-Conversion

For completeness, we manually apply if-conversion (using conditional moves) to branches with small control-dependent regions (individual and nested hammocks). Figure 14 shows that if-conversion increases performance up to 76% and 23% on average, and reduces energy consumption by up to 35% and 16% on average.

Note that there is no overlap between the if-converted and control-flow decoupled applications.

Original Loop

```
1   for ( … ) {
2     index1=index-yoffset-1;                          // 8 instances of this body exist
3     if (waymap[index1].fillnum != fillnum)            // hard-to-predict branch (outer predicate)
4       if (maparp[index1] == 0) {                      // hard-to-predict branch (inner predicate)
5         bound2p[bound2l]=index1;
6         bound2l++;
7         waymap[index1].fillnum=fillnum;               // loop-carried dependency
8         waymap[index1].num=step;
9       }
10  }
```

Decoupled Loops

First Loop

```
1   for ( … ) {
2     index1=index-yoffset-1;
3     pred = (waymap[index1].fillnum != fillnum);        // the outer predicate is computed
4     Push_BQ(pred);                                    // then pushed onto the BQ
5   }
```

Second Loop

```
6   for ( … ) {
7     Branch_on_BQ{                                     // pop the outer predicate
8         index1=index-yoffset-1;
9         output = waymap[index1].fillnum;
10        pred = (output != fillnum) & (maparp[index1] == 0);  // evaluate the overall predicate
11        Push_BQ(pred);                                // push the overall predicate
12        CMOV(output, fillnum, pred);                  // conditional move
13        waymap[index1].fillnum = output;              // always  store
14    }
15    else Push_BQ(0);                                  // needed since we always pop in the 3rd loop
16  }
```

Third Loop

```
17  for ( … ) {
18    Branch_on_BQ{                                     // pop the overall predicate
19        index1=index-yoffset-1;
20        bound2p[bound2l]=index1;
21        bound2l++;
22        waymap[index1].num=step;
23    }
24  }
```

Figure 12: ASTAR source code.

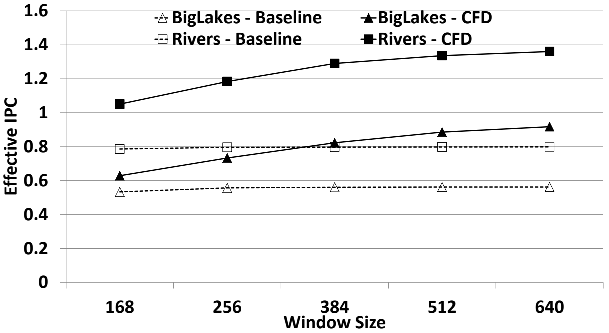
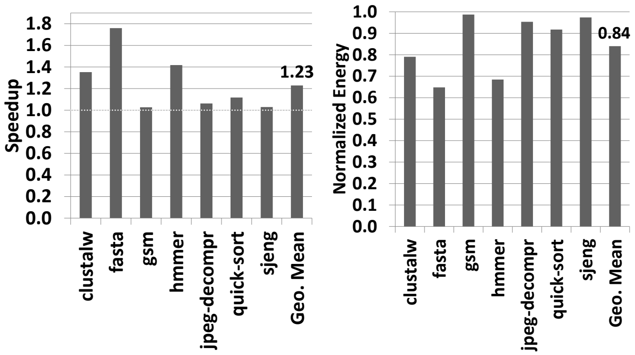


Figure 13: ASTAR: effective IPC of base and CFD binaries as we scale the window size.



(a) Speedup (b) Energy relative to baseline

Figure 14: Impact of if-conversion.

7. Related Work

There has been a lot of work on predication and branch pre-execution. We focus on the most closely related work.

Various ingenious techniques for predication have been proposed, such as: software predication [2], predication using hyperblocks [22], dynamic hammock predication [16], wish branches [15], dynamic predication based on frequently executed paths [14], and predicate prediction [26], to name a few. In this paper, predication (i.e., if-conversion) is a key enabling mechanism for applying CFD to partially separable branches.

CFD resembles branch pre-execution [11, 36, 27, 8, 9]. The key difference is that CFD preserves the simple sequencing model of conventional superscalar processors: in-order instruction fetching of a single thread. This is in contrast with pre-execution which requires thread contexts or cores, and a suite of mechanisms for forking helper threads (careful timing, value prediction, etc.) and coordinating them in relation to the main thread. With CFD, a simplified microarchitecture stems from software/hardware collaboration, simple ISA push/pop rules, and recognition that multiple threads are not required for decoupling.

We now discuss several branch pre-execution solutions in more detail.

Farcy *et. al.* [11] identified backward slices of applicable branches, and used a stride value predictor to provide live-in values to the slices and in this way compute predictions several loop iterations in advance. The technique requires a value predictor and relies on live-in value predictability. CFD does not require either.

Zilles and Sohi [36] proposed pre-executing backward slices of hard-to-predict branches and frequently-missed loads using *speculative slices*. Fork point selection, construction and speculative optimization of slices were done manually. Complex mechanisms are needed to carefully align branch predictions generated by speculative slices with the correct dynamic branch instances. Meanwhile, CFD's Push_BQ/Branch_on_BQ alignment is far simpler, always delivers correct predicates, and has been automated in the compiler.

Roth and Sohi [27] developed a profile-driven compiler to extract data-driven threads (DDTs) to reduce branch and load penalties. The threads are non-speculative and their produced values can be integrated into the main thread via register integration. Branches execute more quickly as a result. Similarly, CFD is non-speculative and automation is demonstrated in this paper. CFD interacts directly with the fetch unit, eliminating the entire branch penalty. It also does not have the microarchitectural complexity of register integration. The closest aspect is the VQ renamer, but the queue-based linking of pushes and pops via physical register mappings is simpler, moreover, it is an optional enhancement for CFD.

Chappell *et al.* [8] proposed Simultaneous Subordinate Microthreading (SSMT) as a general approach for leveraging unused execution capacity to aid the main thread. Originally, programmer-crafted subordinate microthreads were used to implement a large, virtualized two-level branch predictor. Subsequently, an automatic run-time microthread construction mechanism was proposed for pre-executing branches [9].

In the Branch Decoupled Architecture (BDA), proposed by Tyagi *et al.* [33], the fetch unit steers copies of the branch slice to a dedicated core as the unmodified dynamic instruction stream is fetched. Creating the pre-execution slice as main thread instructions are being fetched provides no additional fetch separation between the branch's backward slice and the branch, conflicting with more recent evidence of the need to trigger helper threads further in advance, e.g., Zilles and Sohi [36]. Without fetch separation, the branch must still be predicted and its resolution may be marginally accelerated by a dedicated execution backend for the slice.

Mahlke *et al.* [21] implemented a predicate register file in the fetch stage, a critical advance in facilitating software management of the fetch unit of pipelined processors. The focus of the work, however, was compiler-synthesized branch prediction: synthesizing computation to generate predictions, writing these predictions into the fetch unit's predicate register file, and then having branches reference the predicate registers as *predictions*. The synthesized computation correlates on older register values because the branch's source values are not available by the time the branch is fetched, hence, this is a form of branch prediction. Mahlke *et al.* alluded to the theoretical possibility of truly resolving branches in the fetch unit, and August *et al.* [3] further explored opportunities for such *early-resolved branches*: cases where the existing predicate computation is hoisted early enough for the consuming branch to resolve in the fetch unit. These cases tend to exist in heavily if-converted code such as hyperblocks as these large scheduling regions yield more flexibility for code motion. Quinones *et al.* [26] adapted the predicate register file for an OOO processor, and in so doing resorted to moving it into the rename stage so that it can be renamed. Thus, the renamed predicate register file serves as an overriding branch predictor for the branch predictor in the fetch unit. CFD's branch queue (BQ) is innovative with respect to the above, in several ways: (1) The BQ provides renaming implicitly by allocating new entries at the tail. This allows for hoisting all iterations of a branch's backward slice into CFD's early loop, whereas it is unclear how this can be done with an indexed predicate register file as the index is static. (2) Another advantage is accessing the BTB (to detect Branch_on_BQ instructions) and BQ in parallel, because we always examine the BQ head (Section 3.3.5). In contrast, accessing a predicate register file requires accessing the BTB first, to get the branch's register index, and then accessing the predicate register file.

NSR [5] does not predict branches at all, rather, a branch waits in the fetch stage for an enqueued outcome from the execute stage. To avoid fetch stalls, a few instructions must be scheduled by the programmer or compiler in between the branch and its producer instruction. This is like branch delay slots except that, because the fetch unit can stall, no explicit NOPs need to be inserted when no useful instructions can be scheduled. NSR is a 5-stage in-order pipeline so its static scheduling requirement is of similar complexity to branch delay slot scheduling. CFD-class branches require our "deep" static scheduling technique (for in-order and out-of-order pipelines, alike) which in turn requires CFD's ISA, software, and hardware support.

Decoupled access/execute architectures [30, 4] are alternative implementations of OOO execution, and not a technique for hiding the fetch-to-execute penalty of mispredicted branches. DAE's access and execute streams, which execute on dual cores, each have a subset of the original program's branches. To keep them in sync on the same overall control-flow path, they communicate branch outcomes to each other through queues. However, each core still suffers branch penalties for its subset of branches. Bird *et al.* took DAE a step further and introduced a third core for executing all control-flow instructions, the control processor (CP). CP directs instruction fetching for the other two cores (AP and DP). CP depends on branch conditions calculated in the DP, however. These loss-of-decoupling (LOD) events are equivalent to exposing the fetch-to-execute branch penalty in a modern superscalar processor.

The concept of loop decoupling has been applied in compilers for parallelization. For instance, decoupled software pipelining [25, 34, 12] parallelizes a loop by creating decoupled copies of the loop on two or more cores that cooperate to execute each iteration. All predicates in the backward slices of instructions in the decoupled loops that are not replicated must be communicated. However, predicates are not sent directly to the instruction fetch unit of the other core. Rather, the predicates are forwarded as values through memory or high speed hardware queues and evaluated in the execution stage by a branch instruction.

8. Conclusion

In this paper, we explored the control-flow landscape by characterizing branches with high misprediction contributions in four benchmark suites. We classified branches based on the sizes of their control-dependent regions and the nature of their backward slices (predicate computation), as these two factors give insight into possible solutions. This exercise uncovered an important class of high misprediction contributors, called separable branches. A separable branch has a large control-dependent region, too large for if-conversion to be profitable, and its backward slice does not contain any of the branch's control-dependent instructions or contains just a few. This makes it possible to separate all iterations of the backward slice from all iterations of the branch and its control-dependent region. CFD is a software/hardware collaboration for exploiting separability with low complexity and high efficacy. The loop containing the separable branch is split into two loops (*software*): the first contains only the branch's predicate computation and the second contains the branch and its control-dependent instructions. The first loop communicates branch outcomes to the second loop through an architectural queue (*ISA*). Microarchitecturally, the queue resides in the fetch unit to drive timely, non-speculative fetching or skipping of successive dynamic instances of the control-dependent region (*hardware*).

Measurements of native execution of the four benchmark suites show separable branches are an important class of branches, comparable to the class of branches for which if-conversion is profitable both in terms of number of static branches and MPKI contribution. CFD eradicates mispredictions of separable branches, yielding significant time and energy savings for regions containing them.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback and André Seznec for shepherding the paper. We thank Mark Dechene for developing the simulator used in the study. This research was supported by Intel and NSF grant CCF-0916481. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] K. Albayraktaroglu *et al.*, "Biobench: a benchmark suite of bioinformatics applications," in *Int'l Symp. on Performance Analysis of Systems and Software*, 2005, pp. 182 –188.

[2] J. R. Allen *et al.*, "Conversion of control dependence to data dependence," in *10th Symp. on Principles of Programming Languages*, 1983, pp. 177–189.

[3] D. August *et al.*, "Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results," in *3rd Int'l Symp. on High-Performance Computer Architecture*, 1997, pp. 84–93.

[4] P. L. Bird, A. Rawsthorne, and N. P. Topham, "The effectiveness of decoupling," in *7th Int'l Conf. on Supercomputing*, 1993, pp. 47–56.

[5] E. Brunvand, "The nsr processor," in *26th Hawaii Int'l Conf. on System Sciences*, vol. 1, 1993, pp. 428–435.

[6] B. Burgess *et al.*, "Bobcat: amd's low-power x86 processor," *IEEE Micro*, vol. 31, no. 2, pp. 16–25, 2011.

[7] D. Carmean, "Inside the pentium 4 processor micro-architecture." Presented at Intel Developer Forum, 2000.

[8] R. Chappell *et al.*, "Simultaneous subordinate microthreading (ssmt)," in *26th Int'l Symp. on Computer Architecture*, 1999, pp. 186–195.

[9] R. Chappell *et al.*, "Difficult-path branch prediction using subordinate microthreads," in *29th Int'l Symp. on Comp. Arch.*, 2002, pp. 307–317.

[10] cTuning, "Collective Benchmark," in *http://cTuning.org/cbench*.

[11] A. Farcy *et al.*, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *31st Int'l Symp. on Microarchitecture*, 1998, pp. 59–68.

[12] J. Huang *et al.*, "Decoupled software pipelining creates parallelization opportunities," in *8th Int'l Symp. on Code Generation and Optimization*, 2010, pp. 121–130.

[13] E. Jacobsen, E. Rotenberg, and J. Smith, "Assigning confidence to conditional branch predictions," in *29th Int'l Symp. on Microarchitecture*, 1996, pp. 142–152.

[14] H. Kim *et al.*, "Diverge-merge processor (dmp): dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in *39th Int'l Symp. on Microarchitecture*, 2006, pp. 53–64.

[15] H. Kim *et al.*, "Wish branches: combining conditional branching and predication for adaptive predicated execution," in *38th Int'l Symp. on Microarchitecture*, 2005, pp. 43–54.

[16] A. Klauser *et al.*, "Dynamic hammock predication for non-predicated instruction set architectures," in *7th Int'l Conf. on Parallel Architectures and Compilation Techniques*, 1998, pp. 278–285.

[17] T. Lanier, "Exploring the design of the cortex-a15 processor," 2011.

[18] H. Q. Le *et al.*, "Ibm power6 microarchitecture," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 639–662, 2007.

[19] S. Li *et al.*, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd Int'l Symp. on Microarchitecture*, 2009, pp. 469–480.

[20] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 190–200, Jun. 2005.

[21] S. Mahlke and B. Natarajan, "Compiler synthesized dynamic branch prediction," in *29th Int'l Symp. on Microarch.*, 1996, pp. 153–164.

[22] S. Mahlke *et al.*, "Effective compiler support for predicated execution using the hyperblock," in *25th Int'l Symp. on Microarchitecture*, 1992, pp. 45–54.

[23] O. Mutlu *et al.*, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *9th Int'l Symp. on High-Performance Computer Architecture*, 2003, pp. 129–140.

[24] R. Narayanan *et al.*, "Minebench: a benchmark suite for data mining workloads," in *Int'l Symp. on Workload Characterization*, 2006, pp. 182–188.

[25] G. Ottoni *et al.*, "Automatic thread extraction with decoupled software pipelining," in *38th Int'l Symp. on Microarchitecture*, 2005, pp. 105–118.

[26] E. Quinones, J.-M. Parcerisa, and A. Gonzalez, "Improving branch prediction and predicated execution in out-of-order processors," in *13th Int'l Symp. on High Perf. Computer Architecture*, 2007, pp. 75–84.

[27] A. Roth and G. Sohi, "Speculative data-driven multithreading," in *7th Int'l Symp. on High-Perf. Computer Architecture*, 2001, pp. 37–48.

[28] A. Seznec, "A 64 kbytes isl-tage branch predictor," in *3rd Championship Branch Prediction*, 2011.

[29] A. Seznec, "A new case for the tage branch predictor," in *44th Int'l Symp. on Microarchitecture*, 2011, pp. 117–127.

[30] J. E. Smith, "Decoupled access/execute computer architectures," in *9th Int'l Symp. on Computer Architecture*, 1982, pp. 112–119.

[31] S. T. Srinivasan *et al.*, "Continual flow pipelines," in *11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2004, pp. 107–119.

[32] Standard Performance Evaluation Corporation, "The SPEC CPU 2006 Benchmark Suite," in *http://www.spec.org*.

[33] A. Tyagi, H.-C. Ng, and P. Mohapatra, "Dynamic branch decoupled architecture," in *17th Int'l Conf. on Comp. Design*, 1999, pp. 442–450.

[34] N. Vachharajani *et al.*, "Speculative decoupled software pipelining," in *16th Int'l Conf. on Parallel Architecture and Compilation Techniques*, 2007, pp. 49–59.

[35] B. Valentine, "Introducing sandy bridge." Presented at Intel Developer Forum, San Francisco, 2010.

[36] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *28th Int'l Symp. on Computer Architecture*, 2001, pp. 2–13.