Control-Flow Decoupling: An Approach for Timely, Non-Speculative Branching

Rami Sheikh, *Member, IEEE*, James Tuck, *Member, IEEE*, and Eric Rotenberg, *Senior Member, IEEE*

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. In this paper, we 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 branching. On a microarchitecture configured similar to Intel's Sandy Bridge core, CFD increases performance by up to 55 percent, and reduces energy consumption by up to 49 percent (for CFD regions). Moreover, for some applications, CFD is a necessary catalyst for future complexity-effective large-window architectures to tolerate memory latency.

Index Terms—Microarchitecture, software/hardware codesign, branch prediction, predication, pre-execution, separable branches, isa extensions, instruction level parallelism

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.

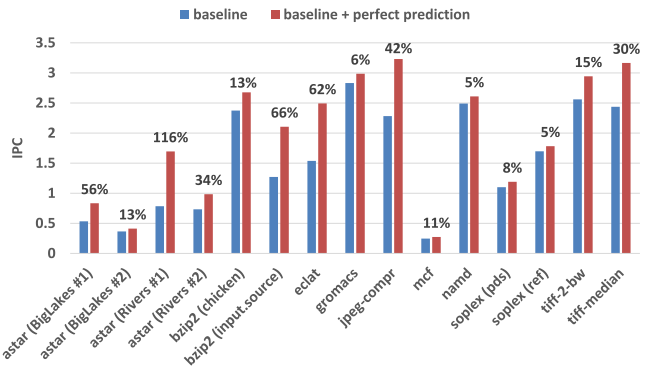
Eliminating branch mispredictions is valuable in this respect. Mispredictions waste both time and energy, firstly, by fetching and executing wrong-path instructions and, secondly, by repairing state before resuming on the correct path. Fig. 1a shows instructions-per-cycle (IPC) for several applications with hard-to-predict branches. The first bar is for our baseline core (refer to Fig. 17a in Section 6) with a state-of-art branch predictor (ISL-TAGE [31], [32]) and the second bar is for the same core with perfect branch prediction. The percentage IPC improvement with perfect branch prediction is shown above the application bars. Speedups with perfect branch prediction range from 1.05 to 2.16. Perfect branch prediction reduces energy consumption by 4 to 64 percent compared to real branch prediction (Fig. 1b).

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 (MLP) with small cycle-critical structures [25], [36]. Their ability to form an effective large window is degraded, however, when a mispredicted branch depends on one of the misses [36]. Fig. 2a shows the breakdown of mispredicted branches with respect to the furthest memory hierarchy level feeding them: L1 cache (*L1*), L2 cache (*L2*), L3 cache (*L3*) or main memory (*MEM*). (*NoData* represents branch mispredictions that are not memory-dependent.) The further away the memory level that feeds the branch, the longer it takes to resolve, and in the case of a misprediction, the more costly the misprediction is. Thus, as the fraction of mispredictions fed by L2, L3 and main memory increases, higher branch prediction accuracy becomes more critical to miss tolerance. This is evident in Fig. 2b, which shows how the IPC of *astar*[1] 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.

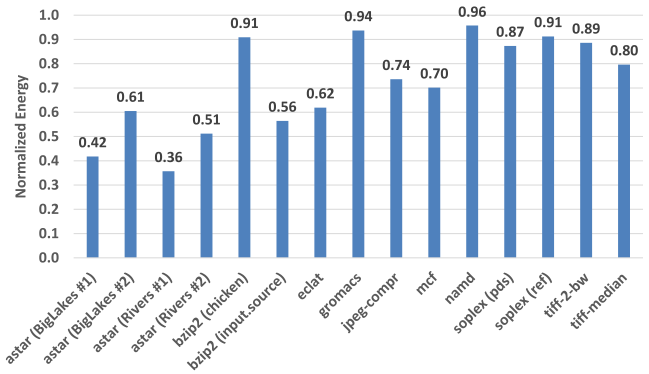
- *R. Sheikh is with Qualcomm Research, Raleigh, NC 27617. E-mail: ralsheik@qti.qualcomm.com.*
- *J. Tuck and E. Rotenberg are with the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh, NC 27695. E-mail: {jtuck, ericro}@ncsu.edu.*

Manuscript received 10 Jan. 2014; revised 24 July 2014; accepted 10 Sept. 2014. Date of publication 2 Oct. 2014; date of current version 10 July 2015. Recommended for acceptance by J. Xue.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2361526

1. We simulate region #1 with the reference input *Rivers*. Refer to the skip distances in Table 3 in Section 7.



(a) IPC



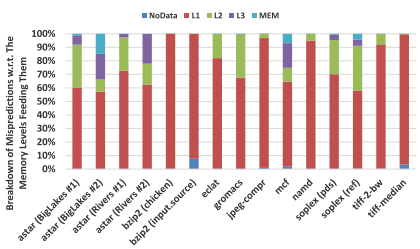
(b) Energy of perfect branch prediction relative to real branch prediction

Fig. 1. Impact of perfect branch prediction.

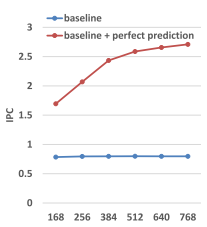
We first explore the current control-flow landscape by characterizing mispredictions in four benchmark suites using a state-of-art predictor. In particular, we classify the control-dependent (CD) regions guarded by hard-to-predict branches. About a third of mispredictions-per-1K-instructions (MPKI) come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

We discover that another third of MPKI comes from what we call *separable* branches. A separable branch has two qualities. First, the branch has a large control-dependent region, not suitable for if-conversion. Second, 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



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



(b) ASTAR(Rivers #1) IPC under different window sizes

Fig. 2. Effect of branch mispredictions on memory latency tolerance.

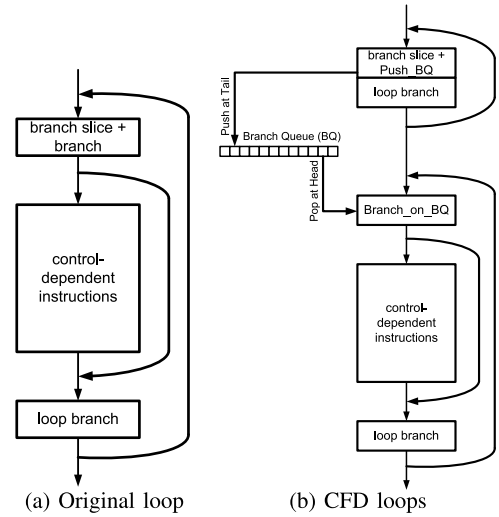


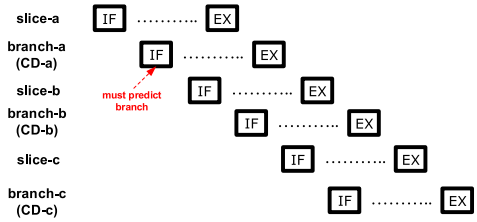
Fig. 3. High-level view of the CFD transformation.

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 *branch slice* (i.e., instructions needed to compute the branch's predicate). This loop generates branch outcomes. A second loop contains the branch and its control-dependent instructions. This loop consumes 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. Fig. 3a shows a high-level view of a totally separable branch within a loop. Fig. 3b shows the loop transformed for CFD.

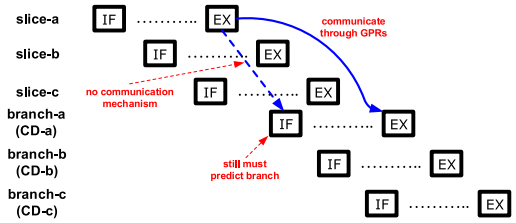
Partially separable branches can also be handled. In this case, the branch's predicate computation depends on some of its control-dependent instructions. This means a copy of the branch and the specific control-dependent instructions must be included in the first loop. Fortunately, this copy of the branch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

The novel idea of CFD targets two problems:

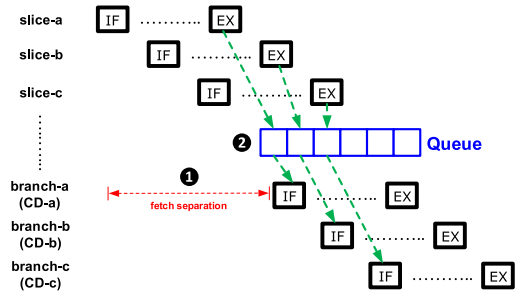
Problem #1. *There is insufficient* fetch separation *between the branch's backward slice (i.e., its predicate computation) and the branch. The branch is fetched very soon after its slice, hence, it is very unlikely that the slice has executed by the time the branch is fetched. The only recourse to avoid stalling the fetch unit is to predict the branch. This problem is illustrated in Fig. 4a, where three loop iterations are shown: iteration-a* (slice-a, branch-a)*, iteration-b* (slice-b, branch-b)*, and iteration-c* (slice-c, branch-c)*. Note that each branch is followed by its control-dependent instructions. As shown, the branch is fetched immediately after its slice and before the slice has executed.*



(a) Problem #1: no fetch separation between branch slice and branch



(b) Problem #2: no mechanism to execute branches in the fetch unit



(c) CFD provides the needed fetch separation (❶) and mechanism to execute branches in the fetch unit (❷)

Fig. 4. Problems addressed by CFD.

CFD addresses this problem by separating the loop containing the branch into two loops: the first loop contains only the branch slice and the second loop contains the branch and its control-dependent instructions. This way, an instance of the branch slice is separated from its corresponding branch by other instances of the branch slice. This provides sufficient fetch separation so that the branch slice can execute before the branch is fetched (generating timely predicates). This is illustrated in Fig. 4c (❶).

Problem #2. *Conventional processors lack support to execute branches in the fetch unit. As shown in Fig. 4b, the slice and branch communicate through general-purpose registers (GPRs), which reside in the execution unit, so the branch must still be predicted, even if sufficient fetch separation is introduced between the branch slice and the branch. Exploiting the timely predicates, to execute the branch in the fetch stage, requires new ISA and hardware support. CFD links the slice to the branch through an architectural queue instead of GPRs, and the hardware implementation of the queue resides in the fetch unit. With fetch separation and explicit predicate communication, CFD effectively executes branches in the fetch stage. This is illustrated in Fig. 4c (❷).*

This paper makes the following main contributions:

1) *CFD* [33], [34]. A software/hardware collaboration technique that exploits branch separability with low complexity and high efficacy. The loop containing the separable branch is split into two loops (*software*):

Example of a separable branch

```
for (i = 0; i < LARGE; i++) {
  if (a[i]) {   // separable branch
    ...
  }
}
```

Example of a separable loop-branch

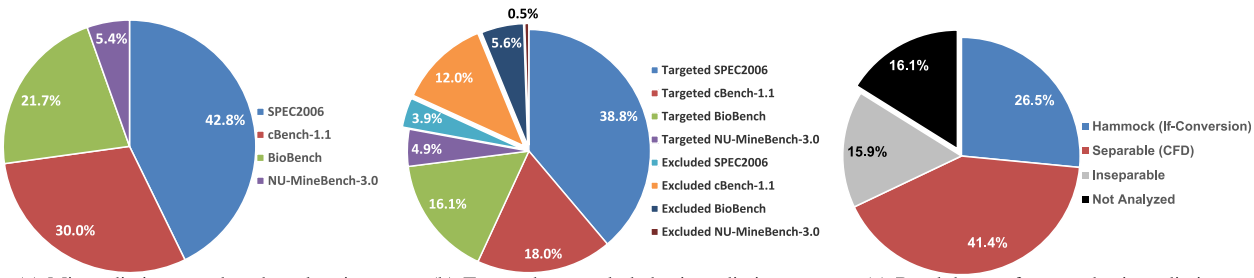
```
for (i = 0; i < LARGE; i++) {
  for (j = 0; j < a[i]; j++) {   // separable loop-branch
    ...
  }
}
```

Fig. 5. Separable branch versus separable loop-branch.

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 branching (*hardware*).

2) *CFD enhancements*. We propose three enhancements for CFD.
   - The first enhancement is a bulk-pop mechanism for removing excess pushes by CFD's first loop. This support is useful when CFD's first loop cannot evaluate early exit conditions present in the original loop.
   - The second enhancement, the value queue (VQ), reduces instruction duplication when a value is needed in both the first and second loops.
   - The third enhancement, the trip-count queue (TQ), allows CFD to be applied to separable loop-branches. Fig. 5 shows the difference between a separable branch and a separable loop-branch. Whereas, a separable branch corresponds to an *if-statement* within a loop, a separable loop-branch corresponds to a *loop-statement* within a loop. Each instance of the loop-statement has a unique iteration count, or "trip-count", which is *a[i]* in the example shown. The data-dependent trip-count, *a[i]*, causes the branch predictor to have trouble predicting when the separable loop-branch exits. The architectural TQ, and modifications to the processor's fetch unit to exploit it, facilitate timely, non-speculative *looping*. To apply CFD to the separable loop-branch: the outer loop is duplicated as before; the first copy of the outer loop generates trip-counts and pushes them onto the TQ; the second copy of the outer loop pops trip-counts from the TQ, and each trip-count is used by the fetch unit to fetch the exact number of iterations of the loop-statement.

3) *DFD*. We propose data-flow decoupling (DFD), a lower-overhead derivative of CFD. Instead of eliminating mispredictions outright, DFD prefetches the misses that feed the mispredictions, thus resolving



(a) Mispredictions per benchmark suite (b) Targeted vs. excluded mispredictions (c) Breakdown of targeted mispredictions

Fig. 6. Breakdown of branch mispredictions.

the mispredictions earlier since the loads that feed them will hit in the cache.

CFD can be applied either manually by the programmer or automatically by the compiler. We implemented and described a *gcc* compiler pass for CFD in our earlier work [33], [34], and demonstrated comparable performance to manual CFD for totally separable branches. Due to limited space, however, we do not discuss our compiler pass in this paper, and all experimental results are based on CFD applied manually. Similarly, several other contributions have been omitted, including but not limited to: extending CFD to support complex scenarios (e.g., multi-level decoupling) [33], applying if-conversion to hammocks [33], [34], and using vector operations to eliminate inseparable branches [33].

On a microarchitecture configured similar to Intel's Sandy Bridge core, CFD increases performance by up to 55 percent and reduces energy consumption by up to 49 percent. 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. DFD increases performance by up to 60 percent and reduces energy consumption by up to 25 percent. On the whole, however, CFD is superior to DFD for two reasons. First, DFD only applies to a subset of the CFD-class applications. Second, as window size is increased, CFD gains scale much better than DFD gains. We conclude that attempting to speed the resolution of mispredicted branches does not compete with eliminating them altogether.

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 basic CFD. Section 4 covers the three CFD enhancements, bulk-pop, VQ, and TQ. Section 5 covers DFD. In Section 6, we describe our evaluation framework and baseline selection process. In Section 7, we present an evaluation of the proposed techniques. In Section 8, we discuss prior related work. We conclude the paper in Section 9.

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* [37] (engineering, scientific, and other workstation type benchmarks), *NU-MineBench-3.0* [26] (data mining), *BioBench* [1] (bioinformatics), and *cBench-1.1* [10] (embedded). All benchmarks[2] are compiled for x86 using gcc with optimization level *-O3* and run to completion using PIN [22]. We wrote a pintool that instantiates a state-of-art branch predictor (winner of CBP3, the third Championship Branch Prediction: 64KB ISL-TAGE [31]) 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.

Fig. 6a shows the relative misprediction contributions of the four benchmark suites. Every benchmark of every suite is included,[3] 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 Fig. 6b. The excluded slice contains (1) benchmarks with misprediction rates less than 2 percent, 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. This work focuses on the targeted slices which, according to Fig. 6b, contribute almost 78 percent of cumulative MPKI in the four benchmark suites. Table 1 lists the targeted benchmarks along with their MPKIs.

2.2 Control-Flow Classification

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

2. 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 *bzip2* we select the ref inputs *input.source* and *chicken*.

3. 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.

TABLE 1
Targeted Applications

Benchmark Suite	Application	MPKI	Benchmark Suite	Application	MPKI
	astar (BigLakes)	10.11		gsm	2.10
	astar (Rivers)	25.98		jpeg-compr	8.17
	bzip2 (chicken)	4.40		jpeg-decompr	2.41
	bzip2 (input.source)	8.16	cBench	quick-sort	4.64
	gobmk	7.17		tiff-2-bw	5.42
SPEC2006	gromacs	1.13		tiff-median	3.60
	mcf	9.06		clustalw	4.25
	namd	1.17	BioBench	fasta	16.90
	sjeng	5.15		hmmer	12.32
	soplex (pds)	6.14			
	soplex (ref)	2.25	MineBench	eclat	10.19

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.

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. An example from *soplex* is shown in Fig. 7a. The branch of interest is at line 3. (This example will be discussed in depth in Section 3.1.) Total separability allows all iterations of the backward slice to be hoisted outside the loop containing the branch, conceptually vectorizing the predicate computation. This is what control-flow decoupling does. The backward slice is *partially separable* if it contains very few of the branch's control-dependent instructions. An example from *astar* is shown in Fig. 7b. The branch of interest is at line 2 and the control-dependent statement in the branch's backward slice is at line 3. 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. Control-flow decoupling will be applied to totally and partially separable branches.

```
1   for ( … ) {
2     x = test[i];
3     if (x < -theeps) {
4       x *= x / penalty_ptr[i];
5       x *= p[i];
6       if (x > best) {
7         best = x;
8         selId = thesolver->id(i);
9       }
10    }
11  }
```

(a) Totally separable branch (from SOPLEX)

```
1   for ( … ) {
2     if (b1arp[j]->nb1ar[i]->fillnum != regfillnum) {
3       b1arp[j]->nb1ar[i]->fillnum=regfillnum;
4       b1arp[j]->nb1ar[i]->waydist=filltact;
5       flend |= (b1arp[j]->nb1ar[i]==rend);
6       b2arp.add(b1arp[j]->nb1ar[i]);
7     }
8   }
```

(b) Partially separable branch (from ASTAR)

Fig. 7. Separable branches.

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 (or in some cases not possible) 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. CFD can not be applied to this class of branches. In our previous work, we proposed using vector operations to eradicate the mispredictions of some of the inseparable branches [33].

Not analyzed. Branches we did not analyze, i.e., branches with small contributions to total mispredictions.

Fig. 6c breaks down the *targeted* mispredictions of Fig. 6b into these four classes. 41.4 percent of the targeted mispredictions can be handled using CFD. 26.5 percent 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 its 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 [33].

3 Control-Flow Decoupling

Fig. 3a 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.

Fig. 3b 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 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. 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 work 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. Next, we discuss ISA, software, and hardware.

3.1 ISA Support and Benchmark Example

ISA support includes (1) an architectural specification of the BQ (size, contents of each entry, and length register), (2) two primary instructions, Push_BQ and Branch_on_BQ, and (3) instructions for saving/restoring the BQ to/from memory on context switches.

The architectural specification of the BQ is as follows. First, the BQ has a specific size. BQ size has implications for software. These are discussed in the next section. Second, 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 not specified in the ISA, i.e., it is not *architectural state*. Therefore, it is not visible to software and is purely design-specific. The microarchitect is free to include additional state to work within the context of a particular pipeline. For example, we include other, microarchitectural state

```
   Original Loop
1  for ( … ) {
2    x = test[i];
3    if (x < -theeps) {                    // hard-to-predict branch
4      x *= x / penalty_ptr[i];
5      x *= p[i];
6      if (x > best) {                     // predictable branch
7        best = x;
8        selId = thesolver->id(i);
9      }
10   }
11 }
   Decoupled Loops
   First Loop
1  for ( … ) {
2    x = test[i];
3    pred = (x < -theeps);                 // the predicate is computed
4    Push_BQ(pred);                        // then pushed onto the BQ
5  }
   Second Loop
6  for ( … ) {
7    Branch_on_BQ{                         // pop the predicate
8      x = test[i];
9      x *= x / penalty_ptr[i];
10     x *= p[i];
11     if (x > best) {
12       best = x;
13       selId = thesolver->id(i);
14     }
15   }
16 }
```

Fig. 8. SOPLEX source code.

in Section 3.3. Third, 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 buffer (uses head and tail pointers) or a shifting buffer (pop shifts all entries to the left and push inserts at index 'length' prior to incrementing 'length', thus, 'BQ[0]' is always the head entry and 'BQ[length-1]' is the tail entry). Thus, at the ISA level, the BQ head and tail are not specified as architectural registers: they pertain to the first and last predicates in the BQ, not their BQ indices (which are implementation-dependent and not visible to software).

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.

Fig. 8 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 percent 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).

Finally, the ISA defines two instructions, Save_BQ and Restore_BQ, to save and restore the BQ state (queue contents and length register) to and from memory. These instructions are required for context-switches. Both Save_BQ and Restore_BQ specify the first address of a chunk of memory large enough to hold the length register plus 'size' predicates. For example, if 'size' is 128, the chunk of memory is 17 bytes: 1 byte for the length register plus 16 bytes for the maximum number of predicates. The first address is specified using a simple addressing mode, such as displacement mode, similar to a load or store instruction. A page fault on any address within the chunk of memory is not prohibited by the ISA. The operating system must either correctly handle this scenario in the context-switch handler or avoid it (e.g., pin the page in physical memory). Save_BQ saves the length register first followed by all predicates between the BQ's head and tail. Restore_BQ restores the length register and 'length' predicates into the BQ. Note that, when the BQ is restored, if it is implemented as a circular buffer with head and tail pointers (at the microarchitecture level), the head and tail pointers are reset to 0 and length-1, respectively, and the saved BQ contents are restored to this range of entries. In fact, the head and tail pointers can be reset arbitrarily, as long as there are 'length' entries between them and these entries are restored from the saved state. This observation is further assurance that only the length register needs to be specified in the ISA.

Adding any new ISA feature, not just CFD, has side-effects. Next, we discuss CFD in the context of three common side-effects of any new ISA feature: impact on future processor generations (obsolescence), impact on security, and impact of having more architectural state (e.g., on simultaneous multithreading (SMT)).

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

Security. CFD, like any new architectural feature, comes with the possibility of new attacks. Many attacks exploit input sequences that are not properly handled by the program (buffer overflows), latent bugs, and so forth. CFD does not change the fact that poorly specified programs (e.g., no bounds checks) or incorrect programs are more vulnerable than well-specified, correct programs. Perhaps, the push/pop ordering rules add another layer of programming complexity that increases the chance of writing an incorrect program. It is beyond the scope of this paper to conceive of (1) specific scenarios of a rarely exercised, if not masked, push/pop ordering violation, and (2) specific exploits of these scenarios by an attacker. We leave this research agenda as future work.

Impact of having more architectural state. CFD introduces new architectural state, namely BQ, TQ and VQ (last two are introduced in Section 4). One impact of more architectural state is longer latency for a context-switch. Processors with simultaneous multi-threading are further impacted because there is more architectural state that needs to be replicated for simultaneous threads.

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 work. We explored multiple solutions but the most straightforward one is loop strip mining [41], where a singly-nested loop is transformed into a doubly-nested one. The outer loop steps through the index set in chunks (of BQ size), and the inner loop steps through each chunk. Then, CFD is applied to the inner loop.

CFD can be applied either manually by the programmer or automatically by the compiler. We implemented and described a *gcc* compiler pass for CFD in our earlier work [33], [34], and demonstrated comparable performance to manual CFD for totally separable branches. Due to limited space, however, we do not discuss our compiler pass in this paper, and all experimental results are based on CFD applied manually.

Applying CFD generally leads to some instruction overhead, even though the original loop body is partitioned between the first and second loops. For example: some values may be needed in both loops (values consumed by both the branch's slice and the branch's control-dependent region); the looping instructions are duplicated; if-converting the branch slice, in the case of a partially separable branch, increases instruction count. Thus, CFD introduces a tradeoff between reducing the number of misspeculated instructions (instructions after mispredicted branches that are fetched and executed, but not ultimately retired) and increasing the number of retired instructions (due to CFD overheads). Whether or not CFD is profitable for a particular separable branch, depends on the misprediction rate and penalty of the branch and the overhead of applying CFD to it. Accordingly, the programmer or compiler must apply the CFD transformation judiciously, leveraging static analysis of the overhead of the CFD-transformed loop, features of the target microarchictecture (e.g., branch predictor or some published proxy), accurate profiling of the branch (misprediction rate and penalty), and iterative compilation [5].

3.3 Hardware Side

This section describes microarchitecture support for CFD. The BQ naturally resides in the instruction fetch unit. In our

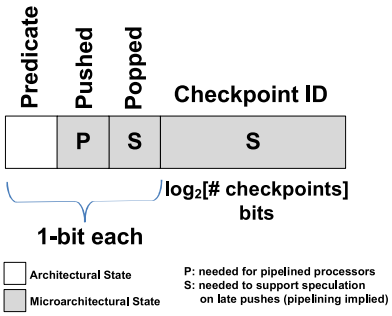


Fig. 9. Fields of a BQ entry.

design, the BQ is implemented as a circular buffer with post-incremented head and tail pointers. In addition to the software-visible predicate bit, each BQ entry has the following microarchitectural state: pushed bit, popped bit, and checkpoint id. The four fields of a BQ entry are shown in Fig. 9. Note that the pushed bit is necessary for CFD to function correctly in a pipelined processor. Meanwhile, the popped bit and checkpoint id are only needed to support speculation on a "late push" (pipelining implied), which will be defined and explained shortly.

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. We explain BQ operation separately for the two possible scenarios: early push and late push.

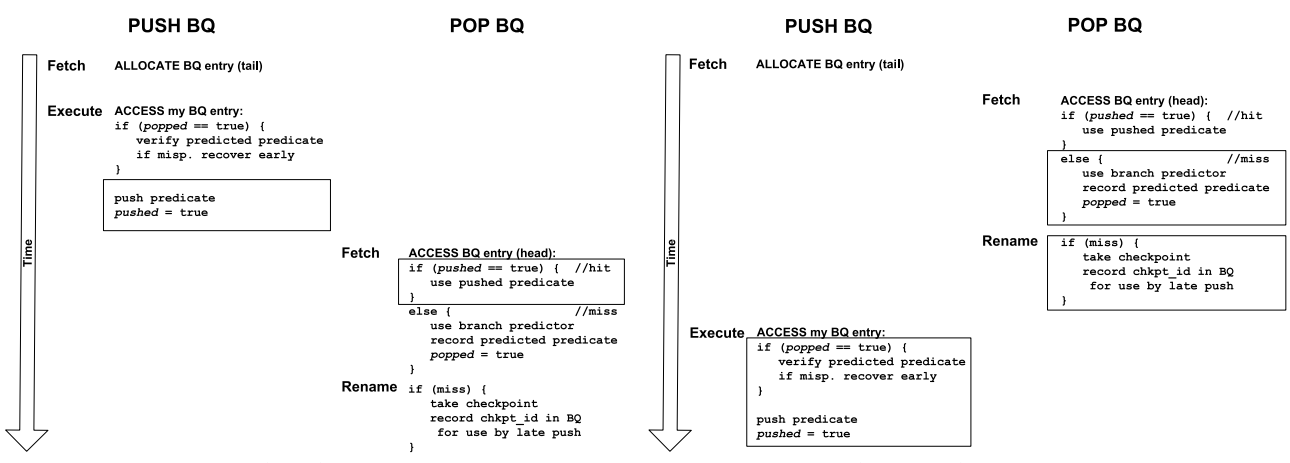
3.3.1 Early Push

The early push scenario is depicted in Fig. 10a. 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. 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 Fig. 10b. 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 6.) 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 percent 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. The late push scenario can also be viewed as a *BQ*



(a) Early push (common) (b) Late push (uncommon)

Fig. 10. BQ operations.

miss, because at the time of fetching the pop instruction, the predicate was not available in the BQ. In the remainder of this paper, we will use the term BQ miss exclusively.

3.3.3 BQ Length

The BQ length (occupancy) is the sum of two components. First, net_push_ctr, which represents 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. Second, pending_push_ctr, which represents 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 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.

- *Preparing for misprediction recovery*. Each branch checkpoint is augmented with a snapshot of the BQ head, tail, and mark pointers (the mark pointer is introduced in Section 4.1).
- *Preparing for exception recovery*. Committed versions of the BQ head, tail, and mark pointers are maintained.

When there is a roll-back, the BQ head, tail, and mark 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 (BTB)

Like all other branch types, Branch_on_BQ is cached in the fetch unit's Branch Target Buffer 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.

4 Control-Flow Decoupling Enhancements

This section discusses three enhancements to CFD.

The first enhancement, comprised of the Mark and Forward instructions, is a bulk-pop mechanism for removing excess pushes by CFD's first loop. This support is useful when CFD's first loop cannot evaluate early exit conditions present in the original loop.

The second enhancement, the value queue, reduces instruction duplication when a value is needed in both the first and second loops.

The third enhancement, the Trip-count Queue, allows CFD to be applied to separable loop-branches.

Note that the first and third enhancements have not been published elsewhere, including the earlier version of this work [34]. The second enhancement (value queue) appeared in the earlier version of this work, but we expand its coverage with a description of ISA support and a benchmark example (ISA and software sides) and an in-depth illustration of its implementation in the rename stage of a superscalar processor (hardware side).

4.1 Support for Nested Breaks: Mark and Forward Instructions

Two new instructions are introduced, namely: Mark, and Forward. The Mark instruction has no register specifiers, and is used to *mark* the BQ tail entry. Similarly, the Forward instruction has no register specifiers, and is used to *bulk-pop* the BQ through to the most recently marked entry. (On a bulk-pop, the length register is decremented by the number of popped entries.) Multiple consecutive Mark instructions are allowed. A Forward instruction merely uses the last Mark. In a CFD-transformed loop, the second loop may have a smaller trip-count than the first loop, due to the original loop having an early exit condition that could not be evaluated in the first loop.[4] Excess pushes from the first loop must be forcibly bulk-popped when the second loop exits early. This is achieved by inserting a Mark instruction and a Forward instruction immediately before and after the second loop, respectively. This will be demonstrated in the detailed case study of *astar* in Section 7.2.

4.2 Reducing Instruction Overhead: Value Queue

In some CFD-transformed loops, 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 and VQ push/pop instructions. We call this optimization CFD+.

4. Typically, this scenario happens when the control-dependent region contains an early exit.

```
                    Decoupled Loops
                        First Loop
1   for ( … ) {
2     x = test[i];
3     Push_VQ(x);                    // the value is pushed onto the VQ
4     pred = (x < -theeps);          // the predicate is computed
5     Push_BQ(pred);                 // then pushed onto the BQ
6   }
                        Second Loop
7   for ( … ) {
8     x = Pop_VQ;                    // pop the value
9     Branch_on_BQ{                  // pop the predicate
10      x *= x / penalty_ptr[i];
11      x *= p[i];
12      if (x > best) {
13        best = x;
14        selId = thesolver->id(i);
15      }
16    }
17  }
```

Fig. 11. Original SOPLEX example, augmented with use of the VQ.

4.2.1 ISA Support and Benchmark Example

ISA support includes (1) specification of the value queue (contents, length register, push/pop ordering rules), (2) two primary instructions for managing the VQ, Push_VQ and Pop_VQ, and (3) context-switch support, Save_VQ and Restore_VQ.

The VQ is similar to the BQ except that, instead of each entry containing a single predicate bit, each entry contains a 32-bit value. The VQ length register is analogous to the BQ length register. The VQ uses the same push/pop ordering rules as before.

Push_VQ has a single source register, which contains the value to be pushed at the tail of the VQ. It does not have an explicit destination register, but it does have an implicit one: the tail entry of the VQ. Pop_VQ has a single destination register, which is where the popped value will be written. It does not have an explicit source register, rather, its source register is implicit: the head entry of the VQ, which contains the value to be popped.

Save_VQ and Restore_VQ have the same behavior as Save_BQ and Restore_BQ, respectively.

An example of using the VQ is shown in Fig. 11. It is the same example from *soplex*, shown previously in Fig. 8, except that it is augmented to communicate the value of *x = test[i]* via the VQ since it is used by both the branch's slice and the branch's control-dependent region.

4.2.2 Hardware Side

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 post-incremented 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. At rename, 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.

Just like the BQ, the VQ renamer state is repaired on a misprediction or exception. Note that nothing special needs to be done to free the physical registers of squashed VQ push instructions. Existing freelist recovery takes care of freeing physical registers of all squashed register-producing instructions.

Fig. 12 illustrates the operation of the VQ renamer in the rename stage of the pipeline, and how it results in leveraging the existing OOO backend (Issue Queue and Physical Register File) as-is. We begin with an example pseudo-assembly code fragment in Fig. 12a of a CFD-transformed loop, with only a few germane instructions shown. The first loop features an add instruction whose result is pushed onto the VQ. We include the add instruction to provide a refresher on how conventional instructions are renamed, and to supply a value to be pushed. The second loop features a pop instruction.

Assuming just two iterations of both loops, Fig. 12d steps through renaming of two instances of the add and push instructions (first loop) followed by two instances of the pop instruction (second loop). The pre-existing renaming structures are the Freelist and Rename Map Table (RMT). The Freelist is the list of physical registers that are currently free, i.e., not allocated to any in-flight instructions or the committed register state. The RMT contains the most recent mappings of logical registers, r0-r31 (assumes the ISA specifies 32 logical registers), to physical registers, p0-p99 (the example assumes 100 physical registers in the Physical Register File). Note that the pre-existing Active List (a.k.a. Reorder Buffer) and Architectural Map Table (AMT), which manage the committing and freeing of physical registers, are not shown, as they operate at the retirement stage with no changes. The only new component, added to the rename stage, is the VQ renamer. Its role is to map the architectural VQ onto the physical register file.

The first add instruction renames its source logical register using the corresponding mapping in the RMT (source r5 renamed to p67). Its destination logical register is renamed by popping a free physical register from the Freelist, and updating the corresponding mapping in the RMT so that its future dependent instructions link to it (destination r5 renamed to p99). The only change to the renaming algorithm is in the handling of the *implicit VQ operands* of the Push_VQ and Pop_VQ instructions: instead of updating or referencing the RMT (respectively), the VQ renamer is updated or referenced for these operands. The first Push_VQ instruction renames its source logical register as usual, by referencing the RMT (source r5 renamed to p99, thereby linking to the add instruction). For its implied destination logical register, the VQ tail, it is allocated a free physical register (p2) from the Freelist, as usual, but this mapping is written to the VQ renamer (destination 'VQ tail' renamed to p2, then VQ tail is incremented) instead of to the RMT. The first Pop_VQ instruction renames its destination logical register as usual, by popping a free physical register from the Freelist and updating the corresponding mapping in the RMT (destination r5 renamed to p51). However, its implied source logical register, the VQ head, is renamed by referencing the head entry of the VQ renamer instead of the RMT (source 'VQ head' renamed

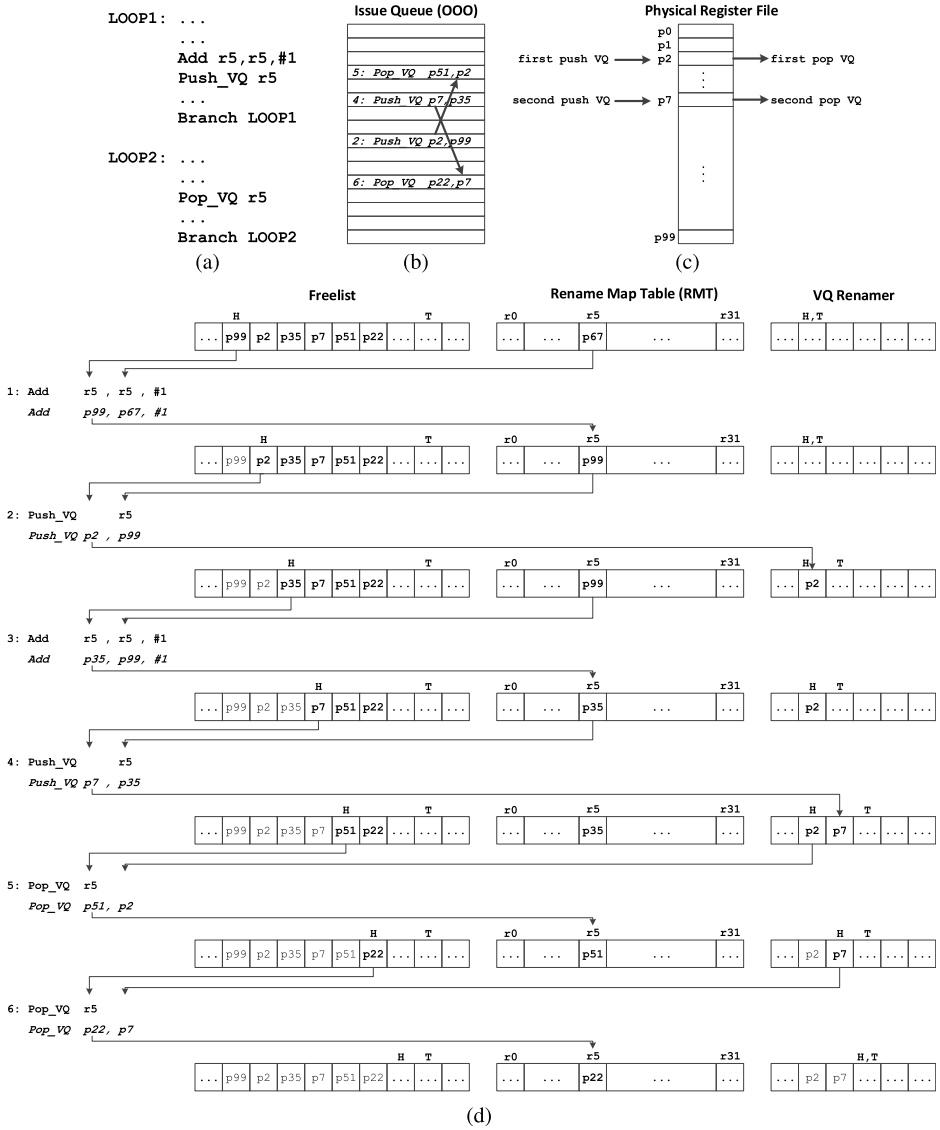


Fig. 12. Illustration of the *VQ renamer* and how it maps the architectural VQ onto the Physical Register File.

to p2, then VQ head is incremented). Notice how the first pop is linked to the first push, via p2, even though there was a second intervening push, which is exactly the desired FIFO queue operation. The second pop is also correctly linked to the second push, via p7.

Finally, linking each push to its corresponding pop via a physical register results in an unmodified backend. Instructions appear arbitrarily ordered in the Issue Queue (scheduler) as instructions issue out-of-order based only on true data dependencies. This is true for the push and pop instructions, too. Correct, serial issuing of producers and consumers is facilitated by their physical register linkages, highlighted with arrows in Fig. 12b. Actual communication of a value between a push and its corresponding pop happens via the Physical Register File, depicted in Fig. 12c. The alternative to our VQ renamer strategy, is to implement a literal value-based VQ separate from the Physical Register File in the backend. This would mean that execution lanes would need to reference either the Physical Register File or the VQ, having ports to both resources, extra MUXes to select either resource, extra bypasses for the VQ, and so forth. Moreover, the Issue Queue's wakeup logic would be heterogeneous, requiring separate wakeup machinery for physical register operands and VQ operands. With just a VQ renamer in the rename stage, we avoid all of this complexity in the already-complex, cycle-time-critical backend of the pipeline.

Save_VQ and Restore_VQ are handled as macro-instructions. The decode stage cracks Save_VQ into multiple pairs of Pop_VQ and store instructions, as many as 'VQ length'. It also generates a store of the VQ length register to memory. Likewise, Restore_VQ is cracked into 'VQ length' pairs of load and Push_VQ instructions, after first restoring the VQ length register from memory.

4.3 Exploiting Separable Loop-Branches: Trip-Count Queue

Even with aggressive loop-branch predictors, some loop-branches remain hard-to-predict and contribute a noticeable fraction of mispredictions. Typically, loop-branches are

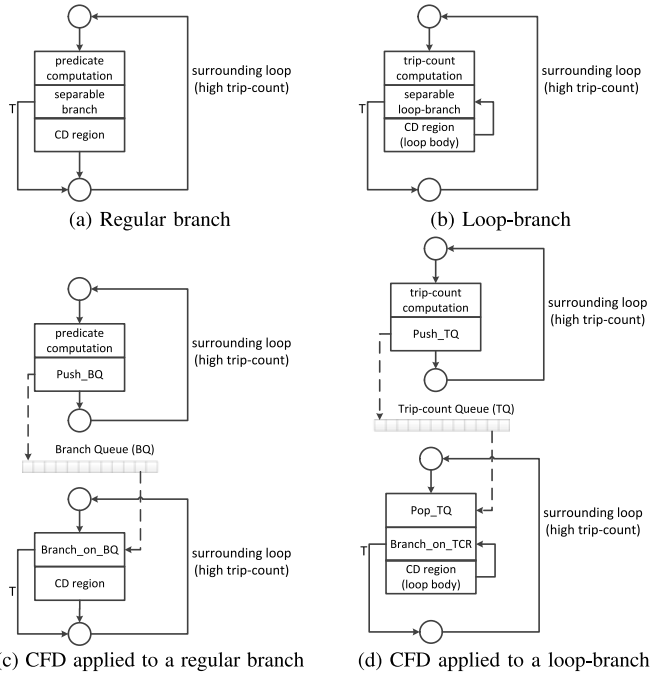


Fig. 13. A generalization of the separability property.

hard-to-predict when they have irregular trip-counts by virtue of the trip-count being data-dependent. Moreover, these hard-to-predict loop-branches become top misprediction contributors when: (1) they have short trip-counts, and (2) they are revisited frequently, e.g., the loop is inside another loop that iterates a lot. We observed some cases where the data-dependent trip-count does not depend on the loop body, i.e., the loop-branch's control-dependent region, therefore, the trip-count computation is separable from the loop-branch and the loop body.

A generalization of the separability property for regular and loop branches is shown in Fig. 13. In both cases, the separable branch/loop-branch is inside of an outer loop. Both cases have a control-dependent region, just of a different nature (forward region versus loop body). The only distinction is predicate computation versus trip-count computation.

CFD can be applied to separable loop-branches as well. In this case, the trip-count computation is separated from the loop-branch and loop body, and the trip-counts are communicated through an architectural trip-count queue. The TQ resides in the fetch unit to drive timely, non-speculative branching. The CFD transformation for regular branches and loop-branches is shown in Figs. 13c and 13d, respectively.

In this section, we highlight how CFD can be extended to support separable loop-branches.

4.3.1 ISA Support and Benchmark Example

ISA support includes (1) the trip-count queue and trip-count register (TCR), and (2) three new instructions, Push_TQ, Pop_TQ, and Branch_on_TCR. The TQ is similar to the BQ except that, instead of each entry containing a single predicate bit, each entry contains a single N-bit trip-count. Push_TQ pushes a trip-count onto the TQ. Pop_TQ pops a trip-count from the TQ and loads it into the TCR. Branch_on_TCR tests the TCR to determine whether to continue or exit the loop. If TCR is not zero, Branch_on_TCR decrements TCR and continues the loop. If TCR is zero, Branch_on_TCR exits the loop.[5]

The same push-pop ordering rules are used for the TQ as for the BQ.

Fig. 14 shows a real example from the benchmark *astar*. Referring to the original code: the loop iterates over an array of arrays of structures. The hard-to-predict loop-branch is at line 3. Its trip-count is data-dependent and we observed it ranges from 0 to 9 (i.e., short trip-count). Although it is data-dependent, the dependence is with the outer loop and not the loop it controls, therefore, this is a separable loop-branch. This branch contributes 12 percent of the benchmark's mispredictions (for *BigLakes* input). Note that the hard-to-predict loop-branch is inside the outer for loop (line 1). Decoupling the trip-count computation is fairly straightforward. The first loop computes the trip-counts and pushes them onto the TQ (line 3). The second loop pops the trip-counts from the TQ (line 7) and conditionally executes the loop instructions (lines 9 through 16), accordingly, using Branch_on_TCR.

4.3.2 Software Side

In this work, we specify a TQ size of 256, and we use loop strip mining when decoupling a long-running outer loop.

4.3.3 Hardware Side

Just like the BQ, the TQ naturally resides in the instruction fetch unit, and it is implemented as a circular buffer. In addition to the software-visible N-bit trip-count, each TQ entry has a pushed bit, shown in Fig. 15a. The TCR also resides in the fetch unit. The TCR tracks how many iterations are left before the loop exits. TCR is loaded with a new trip-count when a Pop_TQ is fetched, and decremented when a Branch_on_TCR is fetched.

Speculating on a TQ miss is more complicated compared to speculating on a BQ miss. The complexity stems from the fact that a single TQ entry corresponds to 2^N instances of Branch_on_TCR. The overhead of maintaining misprediction recovery information, e.g., checkpoint ids for every predicted Branch_on_TCR instruction, can be cumbersome. In our design, we opt to stall the fetch unit on a TQ miss until the Push_TQ executes.

TQ operation, length tracking, and recovery are identical to that of the BQ. Repairing the TCR in the case of branch mispredictions and exceptions requires augmenting each checkpoint with a snapshot of the TCR and maintaining a committed version of the TCR, respectively.

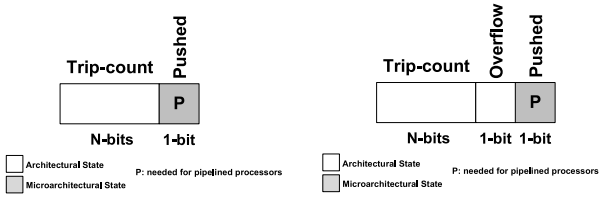
4.3.4 Support for Exceeding Maximum Trip-Count

If the programmer or compiler cannot guarantee that a loop's trip-count is always less than 2^N, then the TQ cannot be used, unless we augment the ISA specification of the TQ to handle the possibility of exceeding the maximum trip-count. Accordingly, we propose the following changes to support loops that may exceed the maximum trip-count:

5. Some ISAs like PowerPC [13] and IA-64 [14] have registers similar to TCR.

```
                    Original Loop
1   for (j=0; j<b1arp.elemqu; j++)
2   {
3     for (i=0; i< b1arp[j]->nb1ar.elemqu; i++)      // hard-to-predict loop-branch
4     {
5       if (b1arp[j]->nb1ar[i]->fillnum!=regfillnum) {
6         b1arp[j]->nb1ar[i]->fillnum=regfillnum;
7         b1arp[j]->nb1ar[i]->waydist=filltact;
8         flend |= (b1arp[j]->nb1ar[i]==rend);
9         b2arp.add(b1arp[j]->nb1ar[i]);
10      }
11    }
12  }
                   Decoupled Loops
                      First Loop
1   for (j=0; j<b1arp.elemqu; j++)
2   {
3     Push_TQ(b1arp[j]->nb1ar.elemqu);          // push trip-count onto the TQ
4   }
                      Second Loop
5   for (j=0; j<b1arp.elemqu; j++)
6   {
7     Pop_TQ();                                 // pop the trip-count
8     for (i=0; Branch_on_TCR; i++)             // predict using trip-count
9     {
10      if (b1arp[j]->nb1ar[i]->fillnum!=regfillnum) {
11        b1arp[j]->nb1ar[i]->fillnum=regfillnum;
12        b1arp[j]->nb1ar[i]->waydist=filltact;
13        flend |= (b1arp[j]->nb1ar[i]==rend);
14        b2arp.add(b1arp[j]->nb1ar[i]);
15      }
16    }
17  }
```

Fig. 14. ASTAR source code.



(a) TQ entry (no overflow support) (b) TQ entry extended to support overflow

Fig. 15. Fields of a TQ entry.

1) Each TQ entry is augmented with a software-visible overflow bit, as shown in Fig. 15 b.
2) The Push_TQ instruction compares the trip-count being pushed to 2^N. If it is less than 2^N, it is written into the TQ and the overflow bit is cleared. If it is not less than 2^N, it is not written into the TQ and the overflow bit is set.
3) A special pop instruction is introduced, called Pop_TQ_and_Branch_on_Overflow. In addition to popping the trip-count, the new instruction specifies a target, via a PC-relative offset, to which control is transferred if the overflow bit is set. Typically, the target is an unmodified version of the loop.

5 Data-Flow Decoupling

In lieu of distancing branch-slices from branches, the first loop can be leveraged to distance load instructions from their dependents. Broadly, this is useful for overlapping misses with each other and with computation. More specifically, it can be used as a lower-overhead alternative to CFD for accelerating mispredicted branches that depend on cache misses. Whereas CFD targets the mispredictions directly, by removing them, one could instead prefetch the misses that feed the mispredictions. This doesn't eliminate mispredictions but it speeds up resolving them as the loads

```
                    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         …
6       }
7   }
                   DFD Loops
                      First Loop
1   for ( … ) {
2     index1=index-yoffset-1;                   // 8 instances of this body exist
3     PREFETCH(waymap[index1].fillnum);         // prefetch the contents of waymap
4     PREFETCH(maparp[index1]);                 // prefetch the contents of maparp
5   }
…                  Second Loop is the Original Loop
```

Fig. 16. DFD example (ASTAR).

that feed the branch will hit in the cache. We call the prefetching application data-flow decoupling. We observed that this can lead to lower overhead in the first loop, due to no longer requiring if-converted control-dependent instructions (for partially separable branches), arbitrarily complex predicate computation, and Push_BQ instructions. Meanwhile, the second loop is the original unmodified loop.

Fig. 16 shows an example from the benchmark *astar*. The original loop contains two nested hard-to-predict branches that are fed by data that frequently miss in the cache (lines 3 and 4). With DFD, we simply precede the original loop by another loop that contains only the load affecting the branch and the load's address slice. This loop prefetches the data and thus speeds up the misprediction resolution in the original loop.

6 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. When applied, CFD and DFD 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 [21], which we augmented with energy accounting for the BQ (CFD, CFD+), VQ renamer (CFD+) and TQ (CFD). Per-access energy for the BQ, VQ renamer and TQ is obtained from CACTI [27] 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 [40]. The baseline core uses the state-of-art ISL-TAGE predictor [31]. 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 [15] versus without) and number of checkpoints (from 0 to 64). We

TABLE 2
Minimum Fetch-to-Execute Latency in Cycles

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

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 eight 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], [7], [19], [20]. We conservatively use 10 cycles for this parameter. We also perform a sensitivity study with this parameter in Section 7.1.

Fig. 17a 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. Fig. 17b shows detailed storage overhead for BQ, VQ renamer, and TQ.

7 Results and Analysis

In this section, we apply and evaluate CFD and DFD. CFD is evaluated for separable branches in Sections 7.1 and 7.2, and for separable loop-branches in Section 7.4. When necessary, we will distinguish between these two cases using "CFD (BQ)" and "CFD(TQ)", respectively. DFD is evaluated in Section 7.3.

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) • **TQ**: 160B (256 5-bit entries)

(a) Baseline core configuration

Branch Queue	**Queue:** 96B (128 6-bit entries)
	Per checkpoint: Snapshot of head, tail, and mark pointers: 3 x 7-bit
	Committed state: Committed version of head, tail, and mark pointers: 3 x 7-bit
	Length register (pending/net): 2 x 8-bit
	Subtotal = 96B + (8 x (3 x 7-bit)) + (3 x 7-bit) + (2 x 8-bit) = 96B + 25.625B = 121.625B
Value Queue Renamer	**Queue:** 128B (128 8-bit entries)
	Per checkpoint: Snapshot of head and tail pointers: 2 x 7-bit
	Committed state: Committed version of head and tail pointers: 2 x 7-bit
	Subtotal = 128B + (8 x (2 x 7-bit)) + (2 x 7-bit) = 128B + 15.75B = 143.75B
Trip-count Queue	**Queue:** 160B (256 5-bit entries)
	Per checkpoint: Snapshot of head and tail pointers: 2 x 8-bit Snapshot of TCR: 4-bit
	Committed state: Committed version of head and tail pointers: 2 x 8-bit Committed version of TCR: 4-bit
	Length register (pending/net): 2 x 9-bit
	Subtotal = 160B + (8 x ((2 x 8-bit) + 4-bit)) + ((2 x 8-bit) + 4-bit) + (2 x 9-bit) = 160B + 24.75B = 184.75B
Overall	**Total = 450.125B**

(b) Detailed storage overheads for BQ, VQ renamer, and TQ

Fig. 17. Baseline configuration and CFD storage overheads.

TABLE 3
CFD(BQ) and DFD Application Skip Distances and Overheads

Application	Skip (B)	Overhead		
		CFD	CFD+	DFD
astar(BigLakes #1)	11.61	1.86	-	1.31
astar(BigLakes #2)	53.99	1.10	-	1.36
astar(Rivers #1)	0.53	1.81	-	1.30
astar(Rivers #2)	164.0	1.11	-	1.34
bzip2(chicken)	0.11	1.02	1.01	-
bzip2(input.source)	0.25	1.27	1.16	-
eclat	7.10	1.28	1.12	-
gromacs	0.74	1.03	1.02	-
jpeg-compr	0.00	1.08	1.06	-
mcf	0.70	1.15	1.14	-
namd	2.17	1.01	-	-
soplex(pds)	9.94	1.02	1.02	1.03
soplex(ref)	49.25	0.90	-	1.03
tiff-2-bw	0.00	1.00	-	-
tiff-median	0.00	1.11	-	-

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 10 M 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 DFD, we simulate as many retired instructions as needed in order to perform the same amount of work as 100 M retired instructions of the unmodified binary.

Tables 3 and 4 show 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 CFD's *soplex*(*ref*), the modified binaries are simulated for more than 100 M retired instructions.

Speedup is calculated as: $cycles_{baseline}/cycles_{modified}$, where $cycles_{baseline}$ is the number of cycles to simulate 100 M instructions of the unmodified binary and $cycles_{modified}$ is the number of cycles to simulate overhead_factor $\times$ 100 M instructions of the modified binary which corresponds to the same simulated region. Effective IPC is calculated as: $instructions_{baseline}/cycles_{scheme}$, where $instructions_{baseline}$ is the number of retired instructions of the unmodified binary

TABLE 4
CFD(TQ) Application Skip Distances and Overheads

Application	Skip (B)	Overhead
astar (BigLakes)	53.99	1.05
astar (Rivers)	164.0	1.05
bzip2 (chicken)	177.0	1.0
bzip2 (input.source)	49.56	1.0

TABLE 5
Details of Modified Code for the CFD(BQ) Applications

Application	File name	Function	Time spent	Loop line	Branch line	Loop strip mining	Communicate values	Promote variables
astar	Way_.cpp (region #1)	makebound2	20% (BigLakes) 47% (Rivers)	57	62-63, 79-80 96-97, 113-114 130-131, 147-148 164-165, 181-182	Y	N	Y
	RegWay_.cpp (region #2)	makebound2	33% (BigLakes) 12% (Rivers)	39	43	N	N	N
bzip2	compress.c	generateMTF Values	30% (chicken) 16% (input.source)	207	214	Y	Y	N
eclat	eclat.cc	get_intersect	46%	205	207, 211	Y	Y	N
gromacs	ns.c	ns5_core	11%	1,503	1,507, 1,508, 1,510	N	Y	N
jpeg-compr	jcdctmgr.c	forward_DCT	83%	231	251	N	Y	N
	jcphuf.c	encode_m-cu_AC_first		488	489			
		encode_mcu_AC_refine		662	663, 686		N	
mcf	pbeampp.c	primal_bea_mpp	39%	165	171	Y	Y	N
namd	ComputeNon-bondedBase.h	ComputeNon-bondedUtil	5%	397	410	Y	N	N
soplex	spxsteeppr.cc	selectLeaveX	5% (pds)	291	295	Y	Y	Y
		selectEnterX	17% (ref)	449	452		N	
tiff-2-bw	tif_lzw.c	LZWDecode	100%	377	411	N	N	N
tiff-median	tiffmedian.c	create_colorcell	100%	725	726	Y	N	N

TABLE 6
Details of Modified Code for the CFD(TQ) Applications

Application	File name	Function	Time spent	Loop line	Loop-branch line	Loop strip mining	Communicate values	Promote variables
astar	RegWay_.cpp	makebound2	33% (BigLakes) 12% (Rivers)	35	39	Y	N	N
bzip2	decompress.c	BZ2_decompress	17% (chicken) 15% (input.source)	474	474	N	N	N

and $cycles_{scheme}$ is the number of cycles to simulate the binary of the given scheme.

Tables 5 and 6 show 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.

7.1 CFD and CFD+

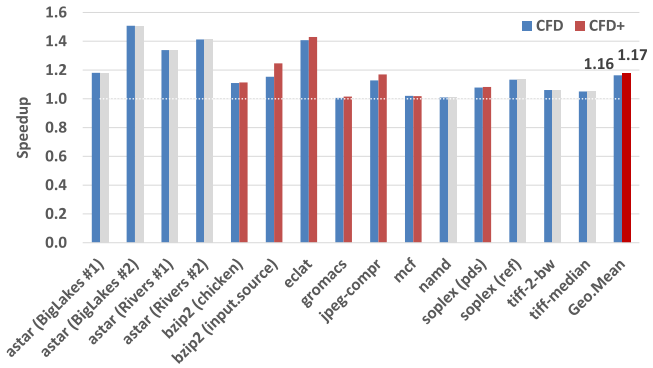
We manually apply then evaluate: CFD and CFD+. Fig. 18a shows that CFD increases performance by up to 51 and 16 percent on average, while CFD+ increases performance by up to 51 and 17 percent on average.[6] Fig. 18b shows that CFD reduces energy consumption by up to 43 and 19 percent on average, while CFD+ reduces energy consumption by up to 43 and 21 percent on average.

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*, region #1) is sped up by 34 percent (s = 1.34) in its CFD region which accounts for 47 percent of its original execution time (f = 0.47); thus, we estimate 14 percent (1.14) speedup overall.

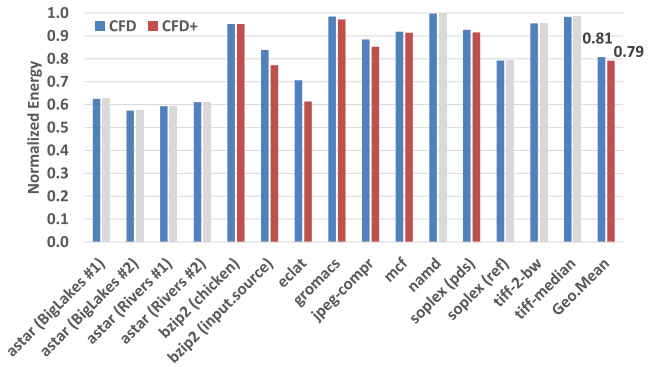
Given the instruction overheads incurred by CFD, and that it always delivers correct predicates, we compare its performance to that of perfect branch prediction. Fig. 19 shows the effective IPC of four configurations: (1) baseline (*Base*), (2) *CFD+*, (3) baseline while perfectly predicting the separable branches (*Base + PerfectCFD*), and (4) baseline with all branches perfectly predicted (*Perfect Prediction*). Three distinct behaviors are observed:

- *Group-1*. CFD underperforms PerfectCFD for *astar* (region #1), *eclat*, *gromacs*, *tiff-2-bw*, and *tiff-median*.
- *Group-2*. CFD matches PerfectCFD for *jpeg-compr*, *mcf*, *namd*, and *soplex*(*pds*).
- *Group-3*. CFD outperforms PerfectCFD for *astar* (region #2), *bzip2*, and *soplex*(*ref*).

For most applications in *Group-1*, CFD underperforms PerfectCFD due to its instruction overheads (shown in Table 3). The only exception to this rule is *tiff-2-bw*. *tiff-2-bw* is the only application where no loop decoupling was performed. Instead, the branch predicate computation was hoisted far ahead within the loop. Unfortunately, when the predicate computation depends on an L1 cache miss, we suffer a BQ miss due to insufficient fetch separation. As a



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



(b) Energy relative to baseline

Fig. 18. Performance and energy impact of CFD.

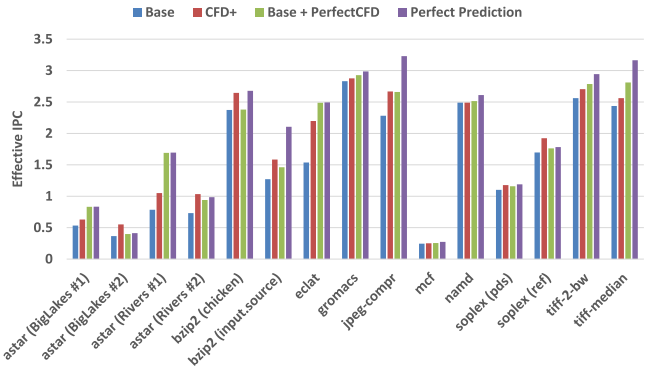
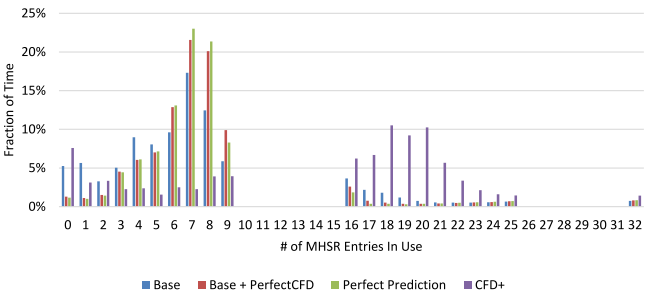
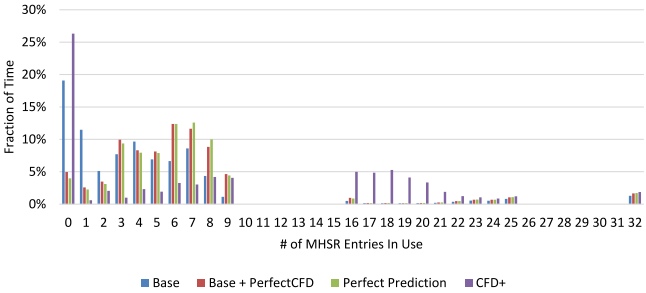


Fig. 19. Effective IPC comparison.



(a) BigLakes #2



(b) Rivers #2

Fig. 20. L1 cache MHSR utilization histograms for ASTAR.

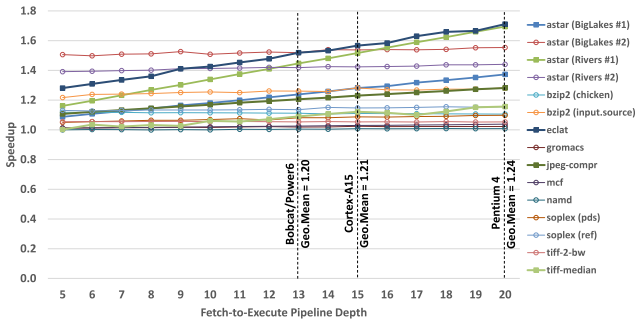
result, the application suffers a relatively high BQ miss rate of 20 percent.[7] So, 20 percent of the branch instances must be predicted, yielding less-than-perfect prediction.

For *Group-2* applications, the CFD instruction overheads are tolerated.

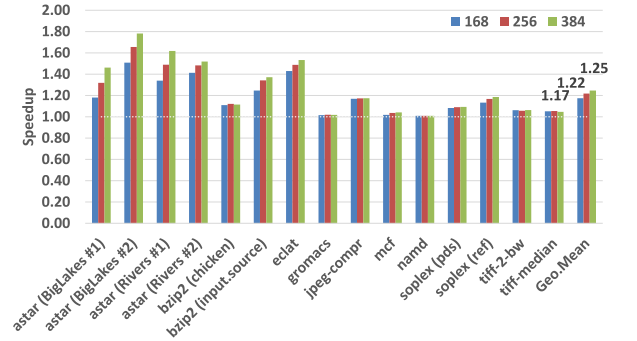
Group-3 applications demonstrate two, very interesting side-effects of CFD. First, CFD can reduce instruction count (compared to baseline) by reducing stack spills and fills. This is the case for *soplex*(*ref*), in which the original loop contains many variables whose live ranges overlap, increasing 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. Second, CFD positively impacts memory-level parallelism by increasing the burstiness of cache misses. By virtue of splitting the original loop into two loops (each loop is smaller than the original), CFD enables more loop iterations to be in the instruction window at any given time, which increases the likelihood of having more concurrent cache misses. Instead of spreading cache misses over N consecutive instruction windows, CFD condenses the misses over M consecutive instruction windows, where $M < N$. In other words, CFD reduces the total number of miss clusters (from N, down to M) by increasing the number of misses in a cluster. Performance is improved because more misses are overlapped within a cluster. To confirm this phenomenon, we studied the utilization of miss handling status registers (MHSRs) at all cache levels. This phenomenon manifests in three ways: first, an increase in the fraction of time when many MHSR entries are in use; second, a decrease in the fraction of time when very few MHSR entries are in use; third, an increase in the fraction of time when zero MHSR entries are in use. Even though this behavior is observed at all cache levels, it is much more obvious in the MHSR utilization histograms of the L1 cache. Fig. 20 shows the L1 cache MHSR utilization histograms for *astar* (region #2), one of the applications in *Group-3*. Notice how CFD+ exhibits a strong bimodal distribution in its histogram. Compared to the other cases without decoupling, CFD+ shows a large fraction of time spent in the zero-utilization bin and high-utilization bins (10 to 32), and low fraction of time in the middle-utilization bins (1-9). This is strong evidence of fewer, denser miss clusters.

Fig. 21a shows speedup with CFD as the minimum fetch-to-execute latency is varied from five to 20 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

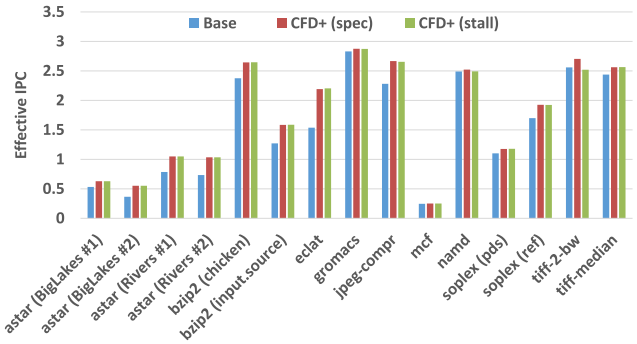
7. All CFD-class applications, except *tiff-2-bw*, have a 99.9 percent BQ hit rate.



(a) Varying the minimum fetch-to-execute latency



(b) Scaling the processor structures



(c) Effective IPC: baseline vs. CFD with/without BQ speculation

Fig. 21. Sensitivity studies.

branch prediction, CFD has the added benefit of exacting performance gains from frequency scaling (i.e., deeper pipelining).

To project the gains of CFD on future processor generations, we evaluate it under larger instruction windows. Fig. 21b shows the projection of CFD gains on two additional configurations labeled in the graph with ROB size.[8] The average performance improvement increases to 25 percent.

CFD-class branches inside loops that do not iterate a lot are more likely to suffer BQ misses due to the insufficient fetch separation between pushes and pops. The CFD-class branches identified in this work are inside loops that iterate a lot, making speculation on a BQ miss less critical. We evaluate CFD with and without speculation support. Fig. 21c shows the effective IPC for three configurations: (1) Baseline (*Base*), (2) CFD with speculation support (*CFD (spec)*), and (3) CFD without speculation support (*CFD (stall)*). In all

8. [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 Fig. 17a in Section 6.

```
                                   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        if (index1==endindex) {                // predictable branch (almost always T)
10          flend=true;
11          return bound2l;
12        }
13      }
14  }
                                   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  }
 6  Mark();                                        // mark the BQ tail pointer
                                     Second Loop
 7  for ( … ) {
 8    Branch_on_BQ{                                // pop the outer predicate
 9      index1=index-yoffset-1;
10      output = waymap[index1].fillnum;
11      pred = (output != fillnum) &  (maparp[index1] == 0);  // evaluate the overall predicate
12      Push_BQ(pred);                             // push the overall predicate
13      CMOV(output, fillnum, pred);               // conditional move
14      waymap[index1].fillnum = output;           // always  store
15      if(index1 == local_endindex & pred) break; // return is replaced with a break
16    }
17    else Push_BQ(0);                             // needed since we always pop in the 3rd loop
18  }
19  Forward();                                     // forward the current BQ head to the mark
                                     Third Loop
20  for ( … ) {
21    Branch_on_BQ{                                // pop the overall predicate
22      index1=index-yoffset-1;
23      bound2p[bound2l]=index1;
24      bound2l++;
25      waymap[index1].num=step;
26      if(index1==local_endindex){
27        flend=true;
28        return bound2l;
29    }
30  }
31  }
```

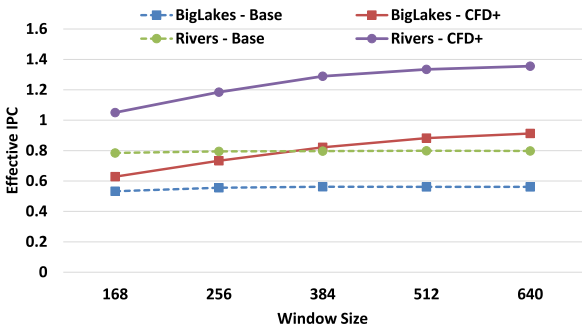
Fig. 22. ASTAR source code (region #1).

applications, except *tiff-2-bw*, there is no major performance loss due to not speculating (i.e., stalling) on a BQ miss. Our expectations are confirmed.

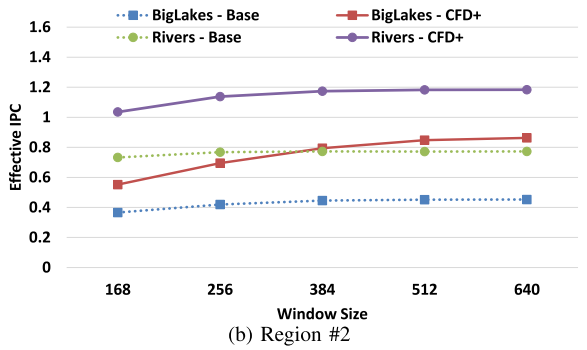
7.2 ASTAR Case Study

One of the most interesting cases we encountered in this work is *astar*, in which CFD is applied to two regions: region #1 and region #2. While both regions pose challenges, we focus the following case study on region #1, as it exhibits more challenging aspects. Fig. 22 shows *astar*'s original and decoupled loops, for region #1. This region has three 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. Third, the control-dependent region contains an early return statement (line 11 of the original loop).

These challenges are handled by CFD, as follows. First, 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. Second, the loop-carried dependency is handled by hoisting and then if-converting the short loop-carried dependency (shown in lines 10, 13 and 14 of the second loop; line 10 is also needed to evaluate the combined predicate). Finally, the return statement is handled by duplicating the



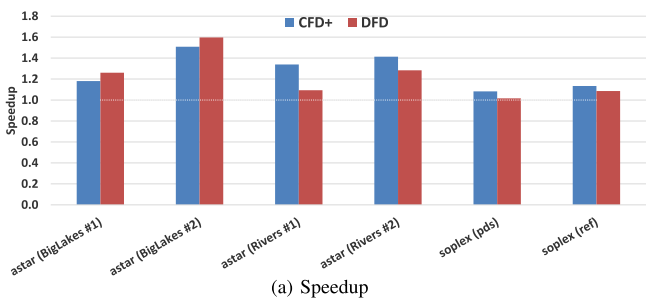
(a) Region #1



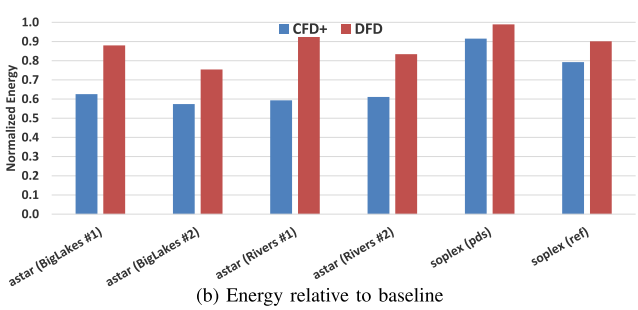
(b) Region #2

Fig. 23. ASTAR: effective IPC as we scale the window size.

condition guarding the return in the second loop and replacing the return statement with an early loop break. This workaround introduces a problem: some of the predicates eagarly pushed by the first loop will not be popped by the second loop. This problem is resolved by using the Mark and Forward instructions introduced in Section 3.1. At the end of the first loop, we mark the tail of the BQ (i.e., the entry following the last predicate pushed by the first loop). At the end of the second loop, we advance the head of the BQ to the previously marked location using the Forward instruction, which ensures that all predicates pushed by the first loop are either popped or skipped by the end of the second loop.

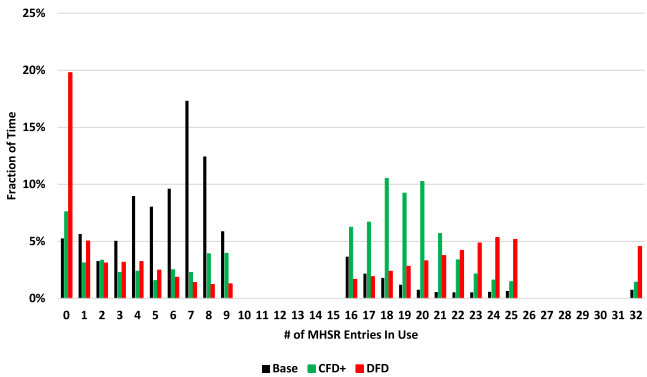


(a) Speedup

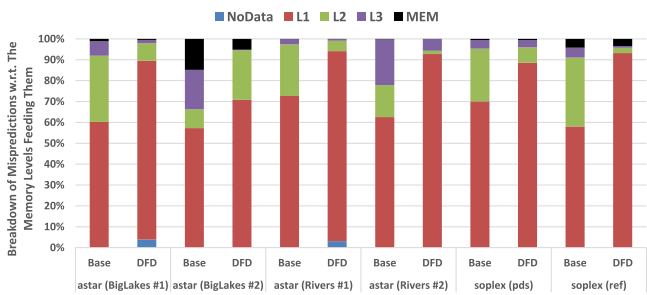


(b) Energy relative to baseline

Fig. 24. Performance and energy impact of CFD and DFD.



(a) ASTAR (BigLakes, region #2)



(b) Breakdown w.r.t. cache level feeding the mispredictions

Fig. 25. DFD's impact on (a) L1 MHSR utilization and (b) misprediction breakdown.

Due to the high percentage of branch mispredictions that are fed by the L2 cache, L3 cache, and main memory, we expect a significant increase in performance gains when we apply CFD to *astar* under large instruction windows. Fig. 23 shows the effective IPC of the unmodified binaries (*Base*), and the *CFD* binaries, as we scale the window size. Our expectations are confirmed for CFD. For example, for the *BigLakes* input and region #2, the speedup increases from 1.51 to 1.91 when window size is increased from 168 to 640.

7.3 DFD

We manually apply then evaluate DFD for CFD-class applications with high L2 and L3 MPKIs. Three applications stand out in terms of misses: *astar*, *soplex* and *mcf*. Even though *mcf* has high L1, L2 and L3 MPKIs, we did not apply DFD to it because the cache misses are encountered outside the CFD region.

Fig. 24 compares the performance and energy impact of CFD and DFD, for *astar* and *soplex*. Fig. 24a shows that DFD increases performance by up to 60 percent.[9] Fig. 24b shows that DFD reduces energy consumption by up to 25 percent. Except for *astar*(*BigLakes*), CFD yields higher speedups than DFD, although DFD performs well. CFD is always significantly more energy-efficient than DFD. Two factors contribute to DFD's superior performance gains in *astar*(*BigLakes*): (1) CFD suffers a

9. We observed at least two scenarios where DFD is profitable. In some cases (e.g., soplex), the address stream is predictable but the absence of an L1 prefetcher left room for improvement. DFD prefetches the data to the L1 cache in this case. In other cases (e.g., astar), the address stream is difficult to predict and can only be accurately identified through pre-execution, and DFD delivers that.

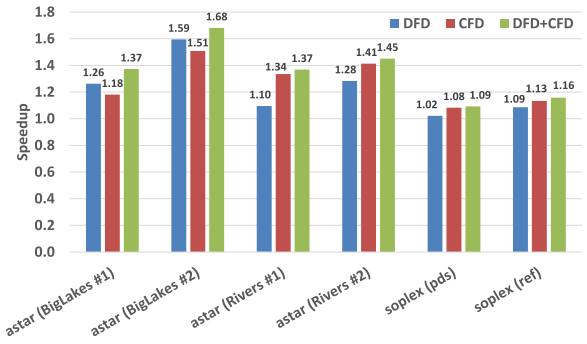


Fig. 26. Performance impact of applying CFD and DFD simultaneously.

significantly higher instruction overhead compared to DFD (1.86 versus 1.31, for region #1), and (2) DFD's first loop is even more compact than CFD's first loop, allowing DFD to have a more aggressive MLP effect. Fig. 25a shows the L1 cache MHSR utilization histograms for *astar* (BigLakes, region #2). While both CFD and DFD exhibit bimodal distributions in their histograms, DFD's is more pronounced. Compared to CFD, DFD shows a larger fraction of time spent in the zero-utilization bin and the eleven highest utilization bins (22 to 32). This is strong evidence of fewer, denser miss clusters in DFD.

We expect DFD to replace mispredictions that depend on distant cache levels with mispredictions that depend on nearby cache levels. Fig. 25b shows the breakdown of mispredicted branches with respect to the furthest memory hierarchy level feeding them, for both baseline and DFD. Fig. 25b confirms that DFD moves the branches' data closer to the core, relative to the baseline.

Interestingly, DFD and CFD can be applied simultaneously: DFD prefetches the data needed for computing the predicates in CFD, allowing the CFD loops to execute faster. Fig. 26 shows the performance improvement when applying DFD only, CFD only, and both.

7.4 Trip-Count Queue

We manually apply then evaluate CFD for separable loop-branches. Fig. 27 shows that CFD(TQ) increases performance by up to 5 percent, and reduces energy consumption by up to 6 percent.

In Section 3, Fig. 14, we showed the original and decoupled loop-branch of *astar*. After eliminating the loop-branch's mispredictions using CFD(TQ), the branch inside the inner loop (line 10 in the second loop) stands out as the main misprediction contributor. Fortunately, this branch is separable and can be targeted with CFD(BQ). Fig. 28a shows that CFD(BQ+TQ) increases performance by up to 55 percent. Fig. 28b shows that CFD(BQ+TQ) reduces energy consumption by up to 49 percent. Interestingly, when both techniques are applied, performance and energy

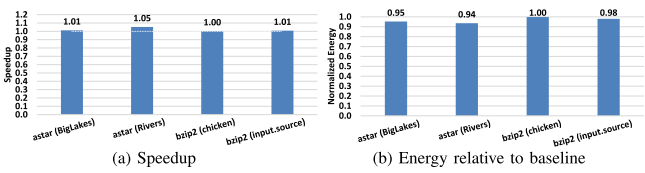


Fig. 27. Performance and energy impact of CFD(TQ).

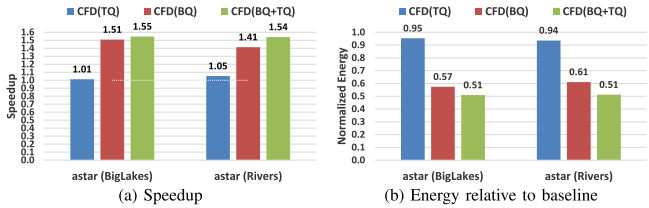


Fig. 28. Performance and energy impact of CFD(BQ, TQ, BQ+TQ).

gains are greater than the sum of the two techniques' individual gains.

8 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 [24], dynamic hammock predication [18], wish branches [17], dynamic predication based on frequently executed paths [16], and predicate prediction [29], to name a few. In this paper, predication (i.e., if-conversion) is a key enabling mechanism for applying control-flow decoupling to partially separable branches.

Control-flow decoupling resembles branch pre-execution [8], [9], [11], [30], [42]. The key difference is that control-flow decoupling 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 control-flow decoupling, 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. Control-flow decoupling does not require either.

Zilles and Sohi [42] 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, control-flow decoupling's push/pop alignment is far simpler, always delivers correct predicates, and has been automated in the compiler [33], [34].

Roth and Sohi [30] 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, control-flow decoupling is non-speculative

and automation is demonstrated in this paper. Control-flow decoupling interacts directly with the fetch unit, eliminating the entire branch penalty. It also does not have the micro-architectural 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. [38], 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 [42]. 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. [23] 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. [29] 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. Control-flow decoupling is innovative with respect to the above, in several ways: (1) The BQ/TQ provide renaming implicitly by allocating new entries at the tail. This allows for hoisting all iterations of a branch's backward slice ahead of the 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/Branch_on_TQ instructions) and BQ/TQ in parallel, because we always examine the head of the queue. 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.

Decoupled access/execute architectures [4], [35] 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 [12], [28], [39] 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.

9 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.

REFERENCES

[1] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung, "BioBench: A benchmark suite of bioinformatics applications," in *Proc. Int. Symp. Perform. Anal. Syst. Softw.*, 2005, pp. 182 –188.

[2] J. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proc. 10th Symp. Principles Program. Languages*, 1983, pp. 177–189.

[3] 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 *Proc. 3rd Int. Symp. High-Perform. Comput. Archit.*, 1997, pp. 84–93.

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

[5] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou, "Iterative compilation in non-linear optimisation space," in *Workshop on Profile and Feedback-Directed Compilation. Organized in conjunction with Int. Conf. Parallel Archit. Compilation Techn.*, 1998.

[6] B. Burgess, B. Cohen, M. Denman, J. Dundas, D. Kaplan, and J. Rupley, "Bobcat: AMD's low-power x86 processor," *IEEE Micro*, vol. 31, no. 2, pp. 16–25, Jan. 2011.

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

[8] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proc. 26th Int. Symp. Comput. Archit.*, 1999, pp. 186–195.

[9] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Difficult-path branch prediction using subordinate microthreads," in *Proc. 29th Int. Symp. Comput. Archit.*, 2002, pp. 307–317.

[10] cTuning, Collective benchmark [Online]. Available: http://cTuning.org/cbench, accessed on 2012.

[11] A. Farcy, O. Temam, R. Espasa, and T. Juan, "Dataflow analysis of branch mispredictions and its application to early resolution of branch outcomes," in *Proc. 31st Int. Symp. Microarchit.*, 1998, pp. 59–68.

[12] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August, "Decoupled software pipelining creates parallelization opportunities," in *Proc. 8th Int. Symp. Code Generation Optimization*, 2010, pp. 121–130.

[13] IBM, The Powerpc Architecture: *A Specification for a New Family of RISC Processors*. San Mateo, CA, USA: Morgan Kaufmann, 1994.

[14] Intel, *IA-64 Application Developers Architecture Guide*, Intel Corporation, Order Number 245188-001, 1999.

[15] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning confidence to conditional branch predictions," in *Proc. 29th Int. Symp. Microarchit.*, 1996, pp. 142–152.

[16] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt, "Diverge-merge processor (DMP): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths," in *Proc. 39th Int. Symp. Microarchit.*, 2006, pp. 53–64.

[17] H. Kim, O. Mutlu, J. Stark, and Y. N. Patt, "Wish branches: Combining conditional branching and predication for adaptive predicated execution," in *Proc. 38th Int. Symp. Microarchit.*, 2005, pp. 43–54.

[18] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *Proc. Int. Conf. Parallel Archit. Compilation Techn.*, 1998, pp. 278–285.

[19] T. Lanier, "Exploring the design of the cortex-a15 processor," presented at ARM Techcon, Santa Clara, CA, USA, 2011.

[20] H. Q. Le, W. J. Starke, S. Fields, F. P. OConnell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden,"IBM POWER6 microarchitecture," *IBM J. Res. Develop.*, vol. 51, no. 6, pp. 639–662, 2007.

[21] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proc. 42nd Int. Symp. Microarchit.*, 2009, pp. 469–480.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2005, pp. 190–200.

[23] S. Mahlke and B. Natarajan, "Compiler synthesized dynamic branch prediction," in *Proc. 29th Int. Symp. Microarchit.*, 1996, pp. 153–164.

[24] 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 *Proc. 25th Int. Symp. Microarchit.*, 1992, pp. 45–54.

[25] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: An alternative to very large instruction windows for out-of-order processors," in *Proc. 9th Int. Symp. High-Perform. Comput. Archit.*, 2003, pp. 129–140.

[26] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary, "MineBench: A benchmark suite for data mining workloads," in *Proc. Int. Symp. Workload Characterization*, 2006, pp. 182–188.

[27] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP Laboratories, Palo Alto, California, USA, Tech. Rep. 85, 2009.

[28] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *Proc. 38th Int. Symp. Microarchit.*, 2005, pp. 105–118.

[29] E. Quinones, J.-M. Parcerisa, and A. Gonzailez, "Improving branch prediction and predicated execution in out-of-order processors," in *Proc. 13th Int. Symp. High Perform. Comput. Archit.*, 2007, pp. 75–84.

[30] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proc. 7th Int. Symp. High Perform. Comput. Archit.*, 2001, pp. 37–48.

[31] A. Seznec, "A 64 kbytes ISL-TAGE branch predictor," in *3rd Championship Branch Prediction*, 2011.

[32] A. Seznec, "A new case for the TAGE branch predictor," in *Proc. 44th Int. Symp. Microarchit.*, 2011, pp. 117–127.

[33] R. Sheikh, "Control-flow decoupling: An approach for timely, non-speculative branching," Ph.D. dissertation, Dept. Elect. Comput. Eng., North Carolina State Univ., NC, USA, 2013.

[34] R. Sheikh, J. Tuck, and E. Rotenberg, "Control-flow decoupling," in *Proc. 45th Int. Symp. Microarchit.*, 2012, pp. 329–340.

[35] J. E. Smith, "Decoupled access/execute computer architectures," in *Proc. 9th Int. Symp. Comput. Archit.*, 1982, pp. 112–119.

[36] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," in *Proc. 11th Int. Conf. Archit. Support Program. Language Operating Syst.*, 2004, pp. 107–119.

[37] Standard Performance Evaluation Corporation, The SPEC CPU 2006 benchmark suite [Online]. Available: http://www.spec.org, accessed on 2010.

[38] A. Tyagi, H.-C. Ng, and P. Mohapatra, "Dynamic branch decoupled architecture," in *Proc. 17th Int. Conf. Comput. Des.*, 1999, pp. 442–450.

[39] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August, "Speculative decoupled software pipelining," in *Proc. 16th Int. Conf. Parallel Archit. Compilation Techn.*, 2007, pp. 49–59.

[40] B. Valentine, "Introducing sandy bridge," presented at Intel Developer Forum, San Francisco, CA, USA, 2010.

[41] M. Wolfe, *High Performance Compilers for Parallel Computing*. Reading, MA, USA: Addison-Wesley, 1995.

[42] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proc. 28th Int. Symp. Comput. Archit.*, 2001, pp. 2–13.

Rami Sheikh received the BS degree in electrical and computer engineering from the Hashemite University in 2006, the MS degree in computer engineering from Jordan University of Science and Technology in 2008, and the PhD degree in computer engineering from North Carolina State University in 2013. He is a senior hardware engineer at Qualcomm CPU Research in Raleigh, NC. His research interests include computer architecture, high-performance microarchitecture, branch prediction, hardware-software co-design, memory hierarchy design, and dynamic binary translation and optimization. He has received several scholarships and awards and is a member of the academic honor society Phi Kappa Phi. He is a member of the IEEE.

James Tuck received the BE degree in computer engineering in 1999 from Vanderbilt University, and the MS degree in electrical engineering in 2004 and the PhD degree in computer science in 2007 both from the University of Illinois Urbana-Champaign. He is an associate professor in the Department of Electrical and Computer Engineering at North Carolina State University. He joined the department in August 2007 as an assistant professor. His research is focused on the design of multicore processor architectures with a focus on hardware/software interactions. He has received two IEEE Micro Top Picks paper awards for his work on multicore processor design. He has served on the organizing and program committees for top conferences in computer architecture and related fields. He is a member of the IEEE.

Eric Rotenberg received the BS degree in electrical engineering in 1991, and the MS and PhD degrees in computer sciences in 1996 and 1999, respectively, from the University of Wisconsin - Madison. He is a professor of electrical and computer engineering at North Carolina State University. From 1992 to 1994, he participated in the design of IBM's AS/400 computer in Rochester, MN. His research is in the area of computer architecture with an emphasis on processor architecture. His group is currently promoting the widespread proliferation of microarchitecturally diverse cores for exploiting diverse instruction-level behavior within and across programs. To this end, his group is developing open-source tools for automatically generating RTL designs of whole superscalar processors (http://people.engr.ncsu.edu/ericro/research/fabscalar.htm). He is a senior member of the IEEE.