Slipstream Processors Revisited: Exploiting Branch Sets

Vinesh Srinivasan
Dep't of Elec. and Comp. Engineering
North Carolina State University
Raleigh, NC, USA
vsriniv3@ncsu.edu

Rangeen Basu Roy Chowdhury
Intel Corporation
Hillsboro, OR, USA
rangeen.basu.roy.chowdhury@intel.com

Eric Rotenberg
Dep't of Elec. and Comp. Engineering
North Carolina State University
Raleigh, NC, USA
ericro@ncsu.edu

***Abstract*—Delinquent branches and loads remain key performance limiters in some applications. One approach to mitigate them is pre-execution. Broadly, there are two classes of pre-execution: one class repeatedly forks small helper threads, each targeting an individual dynamic instance of a delinquent branch or load; the other class begins with two redundant threads in a leader-follower arrangement, and speculatively reduces the leading thread. The objective of this paper is to design a new pre-execution microarchitecture that meets four criteria: (i) retains the simpler coordination of a leader-follower microarchitecture, (ii) is fully automated with just hardware, (iii) targets both branches and loads, (iv) and is effective. We review prior pre-execution proposals and show that none of them meet all four criteria.**

We develop Slipstream 2.0 to meet all four criteria. The key innovation in the space of leader-follower architectures is to remove the *forward control-flow slices* of delinquent branches and loads, from the leading thread. This innovation overcomes key limitations in the only other hardware-only leader-follower prior works: Slipstream and Dual Core Execution (DCE). Slipstream removes backward slices of confident branches to pre-execute unconfident branches, which is ineffective in phases dominated by unconfident branches when branch pre-execution is most needed. DCE is very effective at tolerating cache-missed loads, unless their dependent branches are mispredicted. Removing forward control-flow slices of delinquent branches and delinquent loads enables two firsts, respectively: (1) leader-follower-style branch pre-execution without relying on confident instruction removal, and (2) tolerance of cache-missed loads that feed mispredicted branches.

For SPEC 2006/2017 SimPoints wherein Slipstream 2.0 is auto-enabled, it achieves geomean speedups of 67%, 60%, and 12%, over baseline (one core), Slipstream, and DCE.

***Index Terms*—branch prediction, prefetching, hard-to-predict branch, delinquent load, pre-execution, helper threads, control independence**

I. Introduction

Delinquent branches (frequently mispredicted) and loads (frequently cache-missed) remain major limiters of single-thread performance. Individually, they are bad. They are even worse when they coincide: a cache-missed load feeding a mispredicted branch neutralizes the latency hiding ability of large window processors, as all the instructions fetched in the shadow of the miss are squashed.

Figure 1 shows instructions-per-cycle (IPC) of top-weighted SimPoints from some SPEC 2006 and 2017 benchmarks that exhibit delinquent branches and loads. The baseline core uses a 5.5KB VLDP prefetcher [1] and a 64KB TAGE-SC-L branch predictor [2]. The maximum possible IPC for this 4-wide fetch/retire core is 4 IPC. The figure shows IPCs for (1) the baseline core and (2) the same baseline core with perfect branch prediction and perfect L1 data cache (loads/stores always hit). All of them show more than 2x upper-bound speedup potential.

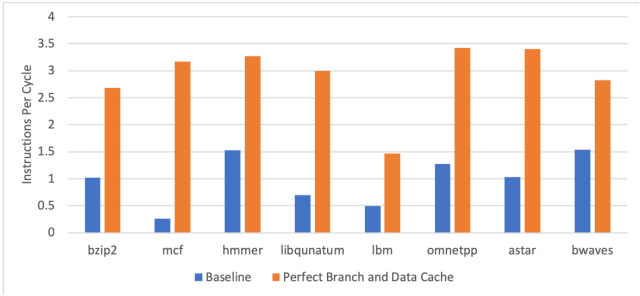


Figure 1: IPC potential in benchmarks with delinquent branches and/or loads.

One approach to mitigate delinquent branches and loads is to exploit some form of pre-execution via helper threads. Helper threads resolve delinquent branches and initiate delinquent loads before the main thread fetches corresponding instances of these instructions. Broadly, there are two classes of pre-execution. One class repeatedly forks small helper threads, each targeting an individual dynamic instance of a delinquent branch or load. Each transient helper thread is the backward slice of instructions leading to the branch/load. The other class begins with two redundant threads in a leader-follower arrangement. The leading thread is speculatively reduced by pruning instructions. Pruning is such that the leading thread still maintains accurate overall control-flow.

The objective of this paper is to design a new pre-execution microarchitecture that meets four criteria. It

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00020

should (i) retain the simpler coordination of a leader-follower microarchitecture as compared to per-dynamic-instance helper threads, (ii) be fully automated with just hardware, (iii) target both branches and loads, and (iv) be effective. We review prior pre-execution proposals and show that none of them meet all four criteria.

Using terminology from one of the first leader-follower processors (Slipstream), we propose Slipstream 2.0. The key innovation in the space of leader-follower architectures is to remove the *forward control-flow slices* of delinquent branches and loads, from the leading thread. The forward control-flow slice of a delinquent branch includes all instructions that are control-dependent on the branch, plus control-independent data-dependent branches with respect to the branch and their control-dependent regions.

This innovation overcomes key limitations in the only other hardware-only leader-follower prior works, Slipstream [3] and Dual Core Execution (DCE) [4]. Slipstream removes backward slices of confidently-predicted branches (replacing the branches with their confident predictions) to pre-execute unconfident branches, which is ineffective in phases dominated by unconfident branches when branch pre-execution is most needed. DCE is very effective at tolerating cache-missed loads, unless their dependent branches are mispredicted. Removing forward control-flow slices of delinquent branches and delinquent loads enables two firsts, respectively: (1) leader-follower-style branch pre-execution without relying on confident instruction removal, and (2) tolerance of cache-missed loads that feed mispredicted branches.

A. Pre-execution, Our Objectives, and Past Works Measured by These Objectives

To review, there are two classes of pre-execution: per-dynamic-instance helper threads and leader-follower redundant threads. The leader-follower class is attractive because coordinating the leader and follower is simple [5]. The leader is always active (although at a higher level it can be enabled only for profitable phases, as developed in this paper) and there is a one-to-one control-flow correspondence between the leader and follower. Thus, it avoids tricky issues of the first class: it does not need careful timing of forking of per-dynamic-instance helper threads and careful alignment of each pre-executed branch outcome to the corresponding branch in the main thread.

The objective of our work is to propose a pre-execution microarchitecture that:

1) retains the simple coordination of a leader-follower microarchitecture,
2) is fully automated with just hardware,
3) targets both branches and loads,
4) and is effective.

To motivate our approach, we characterize past pre-execution proposals in Table I in terms of our four criteria, above.

Slice processor [6], speculative precomputation [7], and continuous runahead [8]. These meet criteria 2 and 4: they are fully automated with just hardware and they are effective at what they target. They do not meet criteria 1 and 3, however: they are not leader-follower and they only target loads.

Speculative data-driven multithreading (DDMT) [9], execution-based prediction using speculative slices [10], and simultaneous subordinate microthreading (SSMT) [11], [12]. These approaches do not meet criteria 1 and 2: they are not leader-follower and they are not fully automated in hardware. For DDMT and speculative slices, backward slices of branches and loads are manually identified, and their trigger (fork) instructions are inserted manually in the main thread. SSMT is fully automated via a compiler: for some industry R&D CPU teams, compiler support for a microarchitecture addition is undesirable as it introduces a dependence between the two that is both difficult to justify and challenging to deploy. These approaches meet criteria 3 and 4: they target both branches and loads, and in principle there are no fundamental limitations to their efficacy as compared to competing approaches.

Slipstream processor [3], [13], [14]. Among the earliest leader-follower architectures, slipstream meets criterion 1. It also meets criterion 2: it is fully automated with hardware. To understand slipstream in the context of criteria 3 and 4, we explain more about how slipstream works. A slipstream processor runs two redundant copies of a program in a leader-follower arrangement on dual superscalar cores (or dual threads in a simultaneous multithreading core) to improve single-thread performance and fault tolerance. The leader (advanced-stream or A-stream) is speculatively reduced by removing confidently predicted branches and their backward slices, which are replaced by the confident predictions. The follower (redundant-stream or R-stream) receives a complete branch history from the A-stream: (i) original predictions for removed confident branches and (ii) pre-executed outcomes for not-removed unconfident branches. We can conclude that, in general, slipstream does not meet criterion 4: in phases with mostly unpredictable branches, where branch pre-execution matters most, slipstream fails to prune enough instructions from the leading thread to be effective. Slipstream's primary pruning criterion is to replace highly confident branches and their backward slices with confident predictions. This criterion is only useful when there is a balanced interleaving of confident and unconfident branches. This conclusion is confirmed in the results section. Slipstream also does not meet criterion 3: it targets branches but not loads. Slipstream's backward slice removal does not stop short at delinquent loads transitively feeding the confident branches, losing out on the opportunity to convert these now-dead loads into non-binding prefetch instructions and exploit high memory level parallelism.

Dual core execution (DCE) [4] . DCE uses dual

Prior work	Criterion 1: leader-follower	Criterion 2: fully automated in hardware	Criterion 3: targets both branches and loads	Criterion 4: effective
Slice processor [6] Speculative precomputation [7] Continuous runahead [8]	no	yes	no (loads only)	yes
DDMT [9] Speculative slices [10] SSMT [11], [12]	no	no (manual or compiler)	yes	yes
Slipstream processor [3], [13], [14]	yes	yes	no (branches only)	no (limited branch pre-execution)
Dual core execution (DCE) [4]	yes	yes	no (loads only)	yes, with caveat (load -> misp. br.)
Decoupled look ahead (DLA) [5], [15], [16]	yes	no (tool)	yes	branches: no (limited branch pre-execution) loads: yes, with caveat (load -> misp. br.)

Table I: Related work analysis.

redundant threads like slipstream and is fully automated in hardware, meeting criteria 1 and 2. It does not prune any instructions in the leading thread, *per se*. Instead, cache-missed loads that would otherwise block retirement in the leading thread for many cycles, and the loads' forward slices, are pseudo-retired (load and its dependent instructions are unstalled and their invalid results discarded). That is, long-latency loads are dynamically converted to non-binding prefetches in the leading thread. DCE does not meet objective 3, however: it only targets loads. Thus, for phases heavy on delinquent branches and light on delinquent loads, DCE offers no speedup. DCE meets objective 4, but with an important caveat: it is highly effective in tolerating long-latency loads, as long as they do not have any mispredicted branches in their forward slices. For a pseudo-retired load, its dependent branches must also be pseudo-retired (owing to the load not forwarding a value), hence, resolution of these miss-dependent branches is deferred to the trailing thread. If the trailing thread detects a mispredicted branch, the leading thread is squashed and restarted from the trailing thread: the latency of the load is not hidden because this latency is transferred to the branch's misprediction penalty. In terms of performance, it's similar to the load simply blocking retirement for the duration of the miss (actually it resembles classic runahead [17] since at least other loads are initiated in the shadow of the squash).

Decoupled look ahead (DLA) [5], [15], [16]. DLA generalizes dual redundant threads by combining concepts from slipstream and dual core execution. Confident branches, as determined by offline profiling, and their backward slices are replaced by unconditional branches (akin to slipstream). Delinquent loads, as determined by offline profiling, are reintroduced as non-binding prefetches if they were initially removed by the confident branch pruning. As a combination of slipstream and DCE, DLA meets criteria 1 and 3: it exemplifies leader-follower and it targets both branches and loads. DLA does not meet objective 2: it uses an offline tool to prune instructions from the binary to generate the leading thread's skeleton. Objective 4 is mixed: (i) like DCE, it is highly effective in targeting delinquent loads that feed predictable branches; (ii) like DCE, it cannot hide the latency of a cache-missed load that feeds a mispredicted branch; (iii) like slipstream, branch pre-execution breaks down when it matters most – DLA cannot achieve sufficient pruning for phases with mostly unpredictable branches.

B. Slipstream 2.0

Inspired by slipstream, our approach begins with dual redundant threads (A-stream, R-stream) in a leader-follower arrangement for simple coordination. Rather than remove backward slices of confident branches in the A-stream, the key idea is to remove *forward control-flow slices* of hard-to-predict pre-executable branches and delinquent loads in the A-stream while still ensuring correct control-flow overall. The forward control-flow slice of a delinquent branch includes all instructions that are control-dependent on the branch, plus control-independent data-dependent branches with respect to the branch and their control-dependent regions. A branch is *pre-executable* if it does not depend on itself: its forward control-flow slice can be removed and the next dynamic instance of the branch will still execute correctly.

We propose a key concept called *branch sets*. A branch set is a list of control-independent data-dependent (CIDD) branches, with respect to a pre-executable branch or a load. Branch sets are important for two reasons. First, a branch is pre-executable if it is not in its own branch set. Second, the branch set describes the forward control-flow slice to be removed.

We propose a new slipstream processor that automatically identifies branch sets. The responsible hardware unit:

1) identifies delinquent branches and loads,

2) identifies branches' probable reconvergent points [18], to delineate branches' control-dependent (CD) regions, and
3) identifies branches'/loads' probable CIDD branches using simple forward poisoning of CD regions and CIDD instructions.

Hard-to-predict branches that are not in their own branch sets are identified for *Delinquent Branch Pre-execution (DBP)*. For DBP, the A-stream's fetch unit:

i. fetches and retains the delinquent branch for pre-execution,
ii. skips over the delinquent branch's CD region, and,
iii. for each branch in the delinquent branch's branch set, fetches then discards the branch and skips over its CD region.

An example of a delinquent branch ① and its forward control-flow slice is shown in Figure 2. Its CD region ② has a potential write to r4 ③. Because branch ④ depends on r4, it is CIDD with respect to the delinquent branch. The CD regions of both branches ① and ④ have potential writes to r5, ⑤ and ⑥. Because branch ⑦ depends on r5, it is also CIDD (directly via branch-①/write-⑤ and transitively via branch-④/write-⑥). The forward control-flow slice of the delinquent branch is comprised of its CD region ②, its branch set ④ and ⑦, and the CD regions ⑧ and ⑨ of its branch set. In order for the A-stream to effectively pre-execute the delinquent branch, it needs to remove its entire forward control-flow slice. This leaves only the black boxes, labeled A, B, C, and D, including the delinquent branch and excluding branches in its branch set. Doing this means that the delinquent branch need not be predicted and any potential misprediction penalty is avoided. The delinquent branch and its entire forward control-flow slice is replaced with a predicate computation. The R-stream receives this predicate computation as a highly accurate branch prediction. The R-stream fetches and executes only the correct CD path of the delinquent branch, penalty-free (except for any local mispredictions of branches nested within the skipped CD region), and locally predicts and resolves its branch set and branch set's CD regions. *Summing up, the R-stream receives a pre-executed outcome for the delinquent branch and the A-stream is insulated from the R-stream's local resolution of deferred dependent gaps.*

Loads that frequently miss in the L1 and L2 caches are identified for *Delinquent Load Prefetching (DLP)*. For DLP, the A-stream's fetch unit:

i. fetches the delinquent load and converts it to a non-binding prefetch instruction, and
ii. for each branch in the delinquent load's branch set, fetches then discards the branch and skips over its CD region.

The R-stream receives all A-stream-executed branch outcomes as accurate predictions, but now any missing control-flow is the responsibility of the R-stream to flesh-out using its branch predictor plus execution. Missing control-flow includes discarded branches from the branch sets as well as branches nested within skipped CD regions. Note that, although there may be localized gaps between branches and their reconvergent points (which delineate CD regions), there is still a one-to-one correspondence in global control-flow between the A-stream and R-stream.

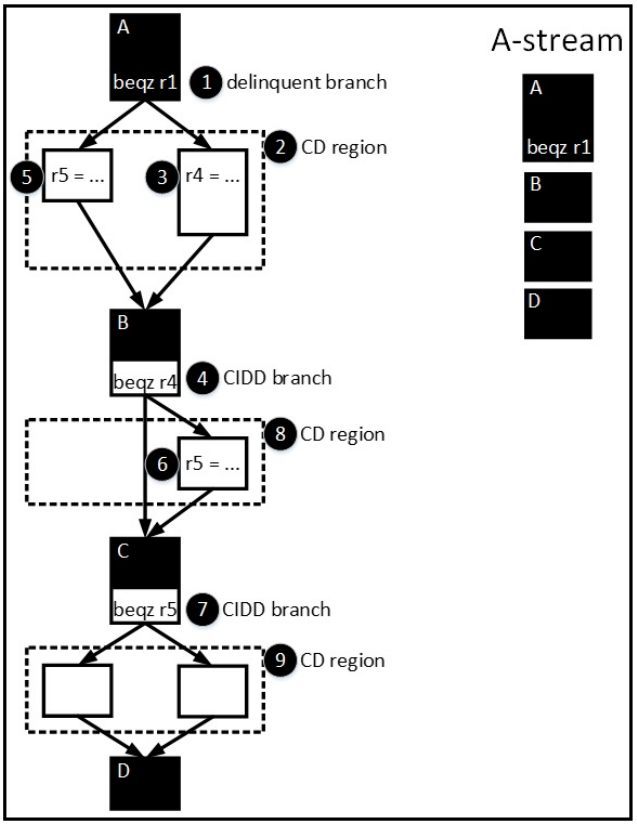


Figure 2: Example of a delinquent branch and its forward control-flow slice.

Slipstream 2.0 meets all four criteria of Section I-A, unlike past pre-execution work: (1) simple coordination of leader-follower style pre-execution, (2) fully automated in hardware, (3) targets both branches and loads, and (4) effective. It effectively pre-executes branches that are pre-executable (not self-dependent), even in regions of mostly unpredictable branches (addressing efficacy weaknesses of original slipstream and DLA). It tolerates long-latency loads that feed mispredicted branches, by insulating the A-stream from the now-localized misprediction recovery in the R-stream (addressing the load efficacy caveat of DCE and DLA).

Another advantage of Slipstream 2.0 is that instruction removal in the A-stream "looks like" traditional branching. Rather than prune individual arbitrary instructions, as slipstream and DLA do, Slipstream 2.0 skips entire CD regions, and *only* CD regions, by converting their branches

to "branch-to-reconvergent-point" instead of "branch-to-taken-target".

Finally, another contribution of this work is a hardware mechanism for enabling/disabling Slipstream 2.0 for phases during which it is profitable/unprofitable, respectively. Thus, Slipstream 2.0 can be used as a microarchitectural turbo-boost performance mode.

Results: First, the turbo-boost feature successfully enables/disables Slipstream 2.0 for benchmark SimPoints according to need. We find that, for most SimPoints, turbo-boost is either enabled (8 out of 25 SimPoints) or disabled (12 out of 25 SimPoints) for almost the entire SimPoint. Second, for the SimPoints where Slipstream 2.0 is enabled, geomean speedups of 67%, 60%, and 12%, are observed over the baseline, slipstream, and DCE, respectively. For benchmarks with primarily delinquent branches, Slipstream 2.0 is 34%, 23%, and 21%, faster than baseline, slipstream, and DCE, respectively. For benchmarks with primarily delinquent loads, Slipstream 2.0 is 84%, 73%, and 9%, faster than baseline, slipstream, and DCE, respectively. Third, Slipstream 2.0 utilizes 4% lesser energy and 43% lesser EDP compared to baseline. Note that these results are for two cores for Slipstream 2.0 versus only a single core for baseline. These results are due to shorter execution time decreasing energy despite redundant core activity, redundant core activity doesn't extend much beyond the two cores' L1 caches, and turbo-boost ensures energy is not wasted in unprofitable phases.

C. Paper Outline

Closely related work was discussed in-depth in Section I-A. Section II describes the Slipstream 2.0 microarchitecture. Section III describes the simulation infrastructure. Section IV presents results, including comparisons with two competing hardware-only leader-follower architectures: slipstream and DCE. Section IV-D discusses energy impact of Slipstream 2.0. Section V concludes the paper.

II. Slipstream Processor 2.0

At a high level, Slipstream 2.0 follows the same microarchitectural blueprint as original Slipstream, so we use similar terms for the added top-level components. The microarchitecture is shown in Figure 3.

The A-stream and R-stream run on two cores. Each core has private L1 instruction and data caches. The L2 and L3 caches are shared between the cores.

The added units and modifications for both Slipstream and Slipstream 2.0 are:

- The A-stream's L1 data cache is speculative, hence, it discards evicted dirty blocks rather than write them back [19]. If the A-stream is squashed and restarted from the R-stream, we follow the invalidate-dirty-block policy for approximately rolling back the A-stream's L1 data cache [19].
- Both use a **Delay Buffer** to communicate pre-executed outcomes from the A-stream to the R-stream. Whereas Slipstream communicated both branch and value outcomes, Slipstream 2.0 only communicates branch outcomes.
- **Instruction-Removal Detector (IR-detector)**: This is the unit that monitors the retired instruction stream and uses criteria to identify instructions that should be removed in future instances. The IR-detector trains the IR-predictor (next bullet) which performs the actual instruction removal at the superscalar's instruction fetch stage. As explained in Section I, Slipstream 2.0 introduces a wholly new IR-detector, with criteria based on delinquent branches, delinquent loads, and their branch sets.
- **Instruction-Removal Predictor (IR-predictor)**: This is the unit that prunes instructions at the instruction fetch stage. Whereas Slipstream used per-dynamic-instance confidence counters for pruning arbitrary instructions in a context-sensitive (global branch history) manner, Slipstream 2.0 uses a much simpler and smaller IR-predictor. It maintains an entry for each static delinquent pre-executable branch, static delinquent load, and static branches in their branch sets.

In the remainder of this section, we focus on the units with new implementations that distinguish Slipstream 2.0: IR-detector (Section II-A), IR-predictor (Section II-B), and Delay Buffer (Section II-C).

A. IR-detector

Figure 4 shows the three subcomponents of the IR-detector. The boxes labeled "Identify Delinquent Branches/Loads" and "Identify Reconvergent Points" are independent support mechanisms that supply information needed by "Branch Set Analysis".

1) Identify Reconvergent Points: This subcomponent is continuously training the reconvergent points of branches as they retire, independent of the other subcomponents. We adapted a reconvergence predictor from the literature [18].

The one-entry Active Reconvergence Table (ART) infers the reconvergent point of one static branch at a time over multiple dynamic instances of the branch. It examines program counters (PCs) of instructions retired after the branch and compares them against the same information from past instances, analogous to maintaining a high water mark. These ongoing comparisons gradually increase confidence in the inferred reconvergent point. The *Conf.* field of the ART is incremented after each instance of the branch. When it saturates, the reconvergent point is updated for that branch in the Reconvergence Predictor Table (RPT), and it moves on to analyzing a different branch that comes along.

The RPT holds potential reconvergent points of all branches. We adapted the RPT to hold up to three different reconvergent points per branch. Each has a confidence counter that is initialized or incremented when that reconvergent point is first added or updated, respectively, by

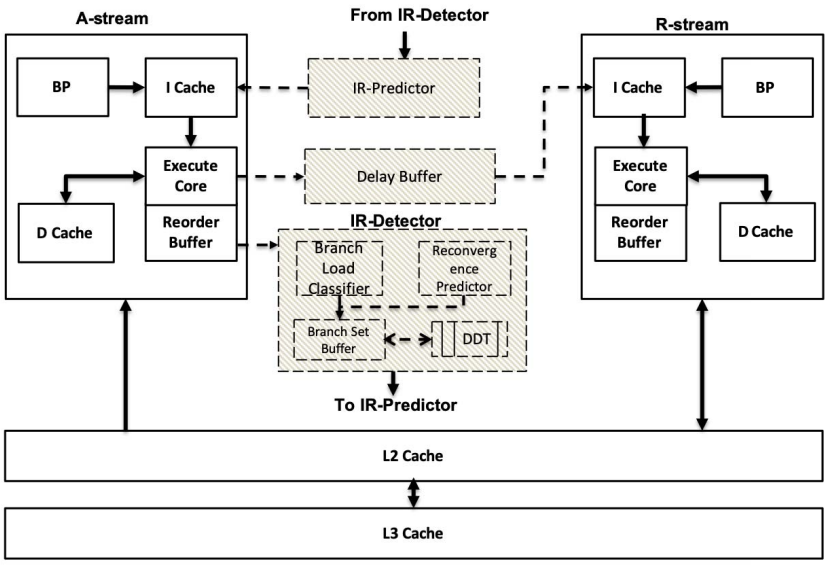


Figure 3: Slipstream Processor 2.0 microarchitecture. Shaded components and dotted lines represent the newly added structures and their connectivity.

the ART. When another subcomponent needs to reference the RPT to get a reconvergent point prediction, the RPT returns the one with the highest confidence. We found that this feature is important to filter-out a distant reconvergent point that may be "truer" in static terms but infrequently exercised in dynamic terms.

The reconvergence predictor (including ART and RPT) requires 1.1 KB of storage, as shown in Figure 4.

2) Identify Delinquent Branches/Loads: Delinquent branches and loads are identified by the Branch/Load Classifier (BLC) shown in Figure 4. The BLC is indexed and tagged by PC. Each entry has a bit indicating whether it is a branch or load. Each entry also has a 16-bit misprediction/miss counter.

The BLC operates on an epoch basis where an epoch is 500K cycles. All counters are cleared at the beginning of an epoch. A branch's or load's counter is incremented each time it is mispredicted or misses in the L2 cache, respectively.

A separate smaller table called "BLC-Max" incrementally maintains a list of the top-8 delinquent loads and branches up to the current point in the epoch. When the BLC is updated, BLC-Max is searched to see if that branch/load is already in the top-8 (matching BLC index found, so just update its counter in BLC-Max) or should knock-off one of the current top-8 (its BLC index does not match any in BLC-Max but its counter is greater than the least counter in BLC-Max). By incrementally maintaining the top-8 list, we avoid serially scanning the BLC for the top 8 at the end of the epoch.

At the end of the epoch, the top-8 delinquent branches and/or loads are sent to the third subcomponent for Branch Set Analysis. The information supplied to Branch Set Analysis is the PC of the branch or load, whether it is a branch or load, and a reconvergent-PC if it is a branch. Thus, at the end of an epoch, the BLC and RPT supply information to kick-off Branch Set Analysis in the next epoch for up to 8 branches/loads. Note that these 8 branches/loads are queued in the unit that does Branch Set Analysis because the analysis is done for one branch or load at a time. After the BLC and RPT kick-off Branch Set Analysis for the next epoch, they return to doing their thing autonomously in the next epoch: the BLC clears all of its misprediction/miss counters and begins anew and the RPT continues training reconvergent points as before.

A 128-entry BLC was found to be sufficient to capture most delinquent loads and branches. The least delinquent branch/load is replaced when there is contention for space in the BLC. Altogether, the BLC and BLC-Max combine for 0.8KB if storage, as shown in Figure 4.

3) Branch Set Analysis: As shown in Figure 4, the Branch Set Buffer (BSB) learns the branch set for one branch or load at a time.

First, the BSB dequeues the next branch or load from its top-8 queue (the queue is not explicitly shown). It writes the branch/load PC and branch reconvergent-PC (for branches only) into the first two fields. In this paper, up to 32 CIDD branches are identified (could do with much fewer in general). Thus, there are 32 CIDD fields in the BSB, each comprised of a valid bit and PC; the valid bits are not explicitly shown but they are initialized to 0. The control-independent data-independent (CIDI) bit (field labeled "CIDI branch") is initialized to 1 for a delinquent branch starting out branch set analysis. This bit remains

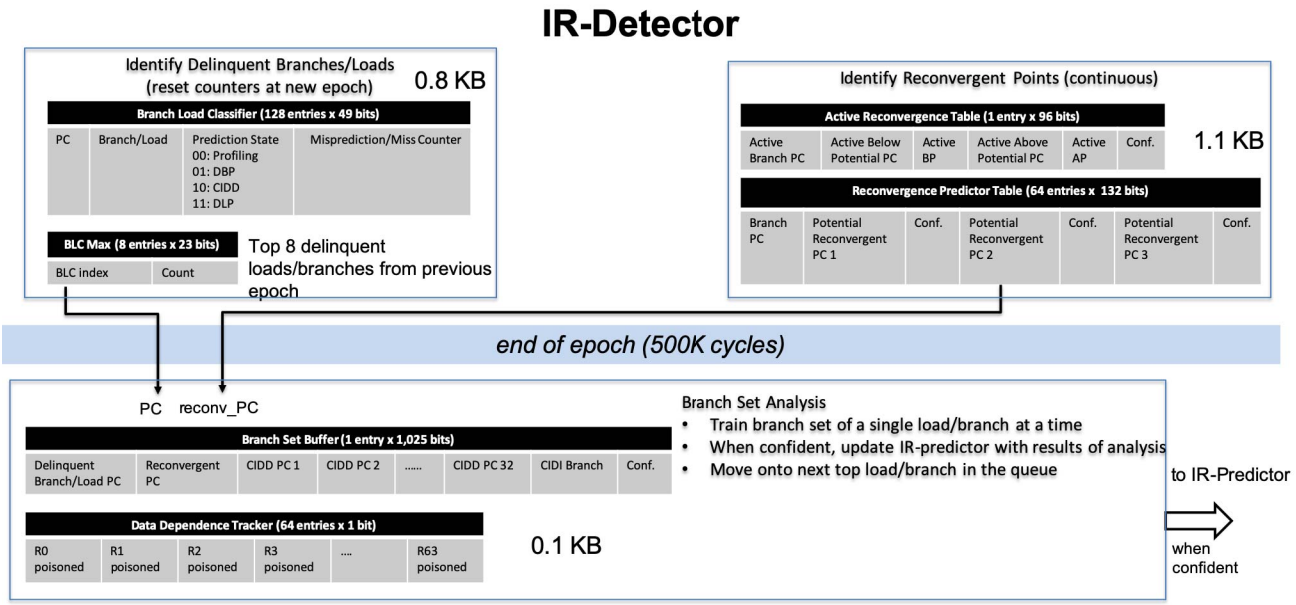


Figure 4: IR-detector.

1 for as long as the delinquent branch is not itself added as a CIDD branch (self-dependent). If it is detected as a CIDD branch with respect to itself, however, the CIDI bit is cleared. At the end of branch set analysis, a delinquent branch with CIDI bit of 1 is deemed pre-executable.

Second, the BSB begins the process of identifying CIDD branches with respect to the delinquent branch/load. This is facilitated by forward poisoning of logical registers influenced by the branch or load, using the Data Dependence Tracker (DDT). The DDT is indexed by logical register specifier (r0-r63 for RISCV ISA) for source/destination registers. Each DDT entry is a single poison bit indicating whether or not that logical register was directly or transitively influenced by the most recent instance of the delinquent branch/load. Each time the delinquent branch/load is retired (as known by PC field in the BSB), the DDT is flash-cleared. For a delinquent branch, each retired instruction in its CD region (as known by reconvergent-PC in the BSB) sets the poison bit of its logical destination register if it has one. That is, any register modified in the CD region is poisoned. After reconvergence, each retired control-independent instruction propagates poison bits from any of its logical source registers to any of its logical destination registers. This aspect identifies CIDD instructions. If a retired control-independent branch has any poisoned source registers, it is added to the CIDD branch list in the BSB if it is not already listed. Moreover, if this CIDD branch is the same as the delinquent branch being analyzed (PC match), the BSB clears the CIDI bit of the delinquent branch (not pre-executable). Finally, if a CIDD branch is detected, the poisoning process temporarily reverts to poisoning all logical destination registers in the CD region of the CIDD branch. This ensures transitive-CIDD branches will also be detected, where a transitive-CIDD branch does not directly depend on the delinquent branch but does depend on one of its other CIDD branches. The overall process is the same for a delinquent load, with the exception that it does not have its own CD region to poison (just CD regions of its CIDD branches as just explained).

Poisoning is restarted fresh (DDT flash-cleared) at each retired instance of the delinquent branch/load in the BSB. The process repeats for multiple instances so that as many paths as possible are explored. The final field in the BSB, labeled "Conf.", is a counter that is incremented for each instance analyzed. When it saturates, the BSB deems the CIDD branch list and the CIDI bit to be accurate.

The final step is to train the IR-predictor when this saturation point is reached. Each IR-predictor entry corresponds to a single branch or load. If it is a branch, it is also annotated as CIDI or CIDD. The IR-predictor is updated by the BSB as follows. The delinquent branch/load is added to (if not already present) or updated in (if already present) the IR-predictor. If it is a delinquent branch, its CIDI bit in the IR-predictor is updated according to its CIDI bit in the BSB. Then, all valid CIDD branches in the BSB (the branch set) are also added to or updated in the IR-predictor; naturally, their CIDI bits are cleared in the IR-predictor.

Altogether, the BSB and DDT have a cost of 0.1 KB.

B. *IR-predictor*

Table II shows the fields of an entry in the IR-predictor. The A-stream's instruction fetch unit indexes the IR-

predictor by PC.

If there is a hit on a load entry, the fetch unit converts the load to a non-binding prefetch.

If there is a hit on a CIDI branch entry, i.e., "pre-execute bit" equal to 1, the fetch unit marks the branch for pre-execution and redirects the fetch PC to its reconvergent-PC. Marking it for pre-execution means resolution of the branch one way or the other does not cause a squash in the A-stream.

Finally, if there is a hit on a CIDD branch entry, i.e., "pre-execute bit" equal to 0, the fetch unit discards the branch instruction and redirects the fetch PC to its reconvergent-PC.

The final field labeled "mispred./miss count" is used to guide replacement of the least-delinquent branch/load when the IR-predictor is using all entries.

We used a 128-entry IR-predictor, which has a storage cost of 1.2 KB.

valid	PC	branch/ load	pre-execute bit (CIDI bit)	reconv. PC	mispred./ miss count
1 bit	30 bits	1 bit	1 bit	30 bits	16 bits

Table II: IR-predictor: 128 entries x 79 bits

.

C. Delay Buffer

Outcomes of A-stream-executed branches are passed to the R-stream through the Delay Buffer. These outcomes are used by the R-stream to predict its corresponding branches, overriding its branch predictor. If a Delay Buffer outcome turns out to be wrong, the R-stream squashes and restarts the A-stream.

Each entry in the Delay Buffer is 3 bits. The first bit indicates whether this branch was executed (1) or discarded (0) by the A-stream. The second bit indicates the outcome, if executed. The third bit indicates whether (1) or not (0) this branch's CD region was skipped in the A-stream. Assuming the first bit is most-significant:

- A branch that is executed by the A-stream normally along with its CD region is encoded as 1x0 (x is outcome). The R-stream can use the outcome, moreover, subsequent outcomes line up, too (no gap in Delay Buffer).
- A pre-executed branch is encoded as 1x1 (x is outcome). The R-stream can use the outcome, but it knows there may be a gap in the Delay Buffer after this branch until its reconvergent-PC is fetched. Thus, the R-stream reverts to its branch predictor for any branches (if any) that are fetched before the reconvergent-PC is fetched.
- A branch that was discarded by the A-stream, because it is CIDD, is encoded as 0-1. The R-stream reverts to its branch predictor for the A-stream-discarded branch because there isn't a valid outcome for it. It also knows there may be a gap in the Delay Buffer after this branch until its reconvergent-PC is fetched. Thus, the R-stream continues using its branch predictor for any branches (if any) that are fetched before the reconvergent-PC is fetched.

For the latter two cases, if the Delay Buffer becomes full before the R-stream fetches the reconvergent-PC, the A-stream is squashed and restarted.

We used a 256-entry Delay Buffer in this paper. This translates to just 0.1KB if we assume the method discussed above, which assumes the R-stream can access the RPT directly for reconvergent-PCs. Another strategy is to include a reconvergent-PC in each Delay Buffer entry, pushed by the A-stream when the third bit is 1. This translates to 1KB. In the next section, we assume the higher storage cost for the Delay Buffer.

D. Proactive vs. Reactive DLP

What we have described for DLP, thus far, is *proactive-DLP*. The IR-predictor directs the A-stream's fetch unit to immediately convert a delinquent load to a prefetch. The IR-detector also trained the IR-predictor such that all of the load's dependent branches are classed as CIDD, hence, never pre-executable. This introduces a dilemma if (1) both the load and branch are delinquent and (2) the branch is eligible for DBP (CIDI with respect to itself). The problem is that proactive-DLP blocks DBP of the branch, even though in hindsight it is pre-executable if the converted load hits.

Reactive-DLP is an alternative that does not leave DBP opportunity on the table when the load hits.

First, the delinquent load is not immediately converted to a prefetch. It is converted to a prefetch at the retire stage, if the load is not resolved when it reaches the head of the reorder buffer.

Second, we modify how a delinquent load in the IR-detector trains the IR-predictor with its CIDD branches: if any of its CIDD branches already exist in the IR-predictor, their CIDI bits are left as-is. Thus, if one of the load's CIDD branches is CIDI with respect to itself, DBP is attempted for it.

With this modification, the load's dependent branch is classed in the IR-predictor as either CIDI (if DBP eligible) or CIDD (if not DBP eligible), and when the branch is fetched in the A-stream, its CD region is skipped in either case. Moreover, in either case, the branch will not cause a recovery in the A-stream (because its CD region was skipped). The only difference is how the branch's execution is handled: if classed as CIDI, its execution is not discarded (push Delay Buffer with encoding 1x1, where x is outcome); if classed as CIDD, its execution is discarded (push Delay Buffer with encoding 0-1).

Finally, if the load is ultimately converted to a prefetch, and if its dependent branch was initially classed as CIDI, we dynamically downgrade the branch from CIDI to CIDD at the point of pushing its encoding on the Delay Buffer. The miss means the branch did not reliably pre-execute.

The downgrade is achieved via a single poison bit per logical register. The converted load sets the poison bit of its logical destination register. A retired instruction sets the poison bit of its logical destination register if any of its logical source registers are poisoned. When the CIDI branch retires, if its source(s) are poisoned, it is downgraded from CIDI to CIDD just prior to pushing its encoding into the Delay Buffer.

E. Storage cost

The total storage cost for the IR-detector, IR-predictor, and Delay Buffer, is 4.2KB.

- IR-detector: 2KB (0.8KB for BLC and BLC-Max; 1.1KB for ART and RPT; 0.1KB for BSB and DDT)
- IR-predictor: 1.2KB
- Delay Buffer (if each entry includes a reconvergent-PC so that R-stream need not access the RPT): 1KB

III. Evaluation

We were able to successfully compile 25 benchmarks, from the SPEC 2006 and 2017 suites, to the RISCV ISA. We generated the top-weighted 100 million instruction SimPoints of these 25 benchmarks.

We use a detailed, cycle-level, execute-at-execute superscalar core simulator that executes the RISCV ISA. It can be configured for a single core or for dual cores in Slipstream 2.0, Slipstream, or Dual Core Execution modes. For all experiments, each superscalar core is configured similarly to the Intel Skylake Desktop Processor (where we have common and major superscalar parameters we set them according to Skylake). This configuration is shown in Table III.

The 22nm technology node provided by McPAT [20] was used to calculate energy for Slipstream 2.0 (two cores) and the baseline (one core). New Slipstream 2.0 components were added to McPAT, with their activity factors gotten from the timing simulator.

IV. Results

A. Slipstream 2.0 versus Baseline

We begin with speedup of Slipstream 2.0 over the baseline (single core). Figure 5 shows speedups of the following configurations over the baseline: Slipstream 2.0 with just Delinquent Branch Pre-execution ("DBP"), baseline with perfect branch prediction ("Perfect BP"), Slipstream 2.0 with just Delinquent Load Prefetching ("DLP"), baseline with perfect data cache ("Perfect DC"), Slipstream 2.0 with both techniques ("DBP+DLP"), and baseline with both perfect ("Perfect BP+DC").

Results are only presented here for the 8 benchmarks for which Slipstream 2.0 is enabled for almost the entire SimPoint by our automatic turbo boost mechanism. The turbo boost mechanism is presented later, in Section IV-C. Section IV-C presents results for all 25 benchmarks, including correlation between branch/load MPKI, turbo boost enabling/disabling, and speedups.

Branch Prediction	BP: 64KB TAGE-SC-L [2], BTB: 4K entries, 4-way, RAS: 32 entries
Hardware Prefetcher	VLDP [1]: 5.5 Kb
L1 I and D caches	32KB, 8-way, 4 cycles
shared L2 cache	256KB, 4-way, 12 cycles
shared L3 cache	8 MB, 16-way, 42 cycles
DRAM	250 cycles
Fetch/Retire Width	4 instr./cycle
Issue/Execute Width	8 instr./cycle
ROB/IQ/LDQ/STQ	224/100/72/72
execution lanes	4 simple ALU, 2 load/store, 2 FP/complex ALU
Fetch-to-Execute Latency	10 cycle
Physical RF	288
Checkpoints	32, OoO reclamation
Slipstream	Delay buffer size: 256 entries

Table III: Core configuration with parameters modeled after Intel Skylake Desktop Processor, for in-common major superscalar parameters.

1) DBP: Benchmarks with high branch MPKI are evident from the speedup of "Perfect BP": bzip, astar, hmmer, and mcf. These four benchmarks achieve 17% to 28% speedup with DBP. There is still a gap between DBP and "Perfect BP" because not all mispredictions can be pre-executed. Table IV breaks down mispredictions into three types. For example, for astar:

- 53% of mispredictions were pre-executed because they originated from pre-executable (CIDI) branches, such as branch-1 in Figure 6 (line 7). These otherwise-mispredicted branches were resolved penalty-free in the A-stream.
- 29% of mispredictions originated from *hypothetically* pre-executable (CIDI) branches, but they were nested inside the skipped CD regions of the other pre-executed branches, above. These mispredictions were deferred and resolved with a local squash penalty in the R-stream, although the A-stream is not squashed. Branch-2 in Figure 6 (line 8) is an example of a pre-executable branch, that was not pre-executed, due to being in the skipped CD region of the pre-executed branch-1.
- 18% of mispredictions originated from CIDD branches. CIDD branches are not pre-executable because the next dynamic instance depends on the CD region of the previous dynamic instance; so the CD region cannot be skipped in the A-stream, hence, a mispredicted instance is resolved with a penalty in the A-stream. We argue that helper threads, in general, cannot help this very serial class of code: a branch that depends on itself.

In the case of astar, another reason for the gap w.r.t. "Perfect BP" is that branch 1 is not purely CIDI with respect to the occasional loop-carried memory dependency with the store at line 13 of Figure 6.

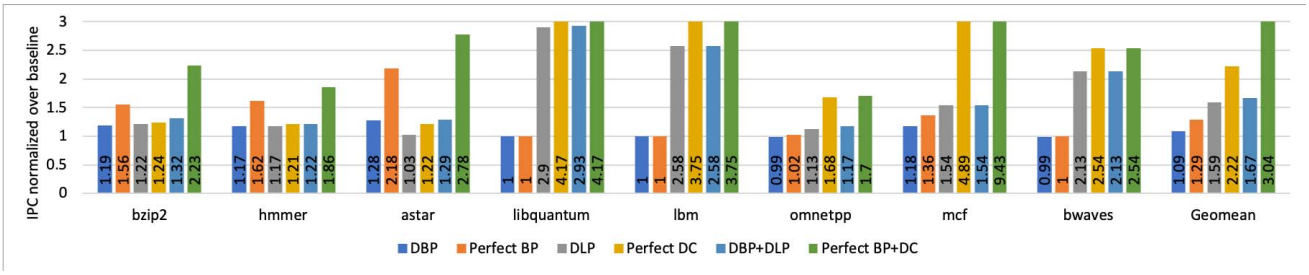


Figure 5: Speedups of Slipstream 2.0 (DBP, DLP, DBP+DLP) over baseline. Baseline with perfect branch prediction and/or data cache also shown as a gauge.

misprediction type	astar	bzip	hmmer	mcf
CIDI misprediction, pre-executed	53%	42%	71%	75%
CIDI misprediction, not pre-executed	29%	29%	3%	0%
CIDD misprediction	18%	29%	26%	25%

Table IV: Breakdown of misp. according to branch type.

```
 1:  bound2l=0;
 2:  for (i=0; i<bound1l; i++)
 3:  {
 4:    index=bound1p[i];
 5:                                        branch 1
 6:    index1=index-yoffset-1;             <-
 7:    if (waymap[index1].fillnum!=fillnum)
 8:      if (maparp[index1]==0)            <- branch 2
 9:      {
10:        bound2p[bound2l]=index1;
11:        bound2l++;
12:
13:        waymap[index1].fillnum=fillnum;
14:        waymap[index1].num=step;
15:
16:        if (index1==endindex)
17:        {
18:          flend=true;
19:          return bound2l;
20:        }
21:      }
22:    ... Above nested-if template repeats
23:    ... 7 times for different index1's.
24:  }  // end for loop
```

Figure 6: Astar code fragment.

2) DLP: DLP benefits benchmarks suffering primarily from L2/L3 cache misses, which are evident from the speedup of "Perfect DC". Libquantum, lbm, omnetpp, mcf, and bwaves, see speedups ranging from 13% to 2.9x. Among these, mcf is notable in having many branch mispredictions that depend on cache-missed loads. Slipstream 2.0 insulates the A-stream from miss-dependent mispredictions that resolve in the R-stream (*i.e.*, A-stream is not restarted). Without this effect, mcf's speedup would decrease from 1.54 (DLP) to 1.35 (as measured for DCE in Sec. IV-B). This effect is also evident in benchmarks with a milder delinquent load problem and high branch MPKI: bzip2 and hmmer get 22% and 17% improvement over baseline with DLP, respectively, which is quite close to "Perfect DC" for them (24% and 21%, respectively).

3) DBP+DLP: All benchmarks show speedups when DBP and reactive-DLP (§II-D) are combined, and the combination matches or exceeds DBP or DLP alone. DBP+DLP achieves a geometric mean speedup of 67%.

B. Comparisons w/ Slipstream 1.0 and DCE

Classic slipstream's instruction removal rate is very low for applications with high branch MPKI. For example, for astar, the A-stream's retired instruction count is reduced by only 4%. Bzip2 and hmmer are similar. The A-stream pays the penalties of all mispredictions – always the case with classic slipstream – and has negligible instruction removal to counterbalance the penalties. As a result, Slipstream 1.0 achieves no speedup over the baseline for high MPKI benchmarks, as seen in Figure 7. DBP overcomes this by identifying CIDI branches to pre-execute in the A-stream: the A-stream is not slowed by any instances of these branches by way of skipping their CD regions.

Classic slipstream achieves 11% to 14% speedup for libquantum, mcf, and omnetpp. The branches in these benchmarks are highly confident and there is decent instruction removal (Fig. 8a) and few A-stream restarts (Fig. 8b). Even though some delinquent loads are transitively removed from the A-stream, owing to classic slipstream's back-propagation, some loads still execute in the A-stream thus achieving some prefetching effect. In contrast, DLP ensures that all delinquent loads are converted to prefetches.

While bzip2, hmmer, astar, and mcf are notable for high branch MPKIs, they also have non-negligible cache misses. From Figure 7, DLP alone achieves speedups of 1.22, 1.17, 1.03, and 1.54, respectively, whereas DCE achieves speedups of 1.03, 1.13, 0.95 (slowdown), and 1.35, respectively. Both DLP and DCE convert delinquent loads to prefetches in the A-stream. Unlike DLP, DCE does not remove the loads' forward control-flow slices and instead relies on the loads' dependent branches to be predicted correctly. Thus, DCE must rollback the A-stream if a dependent branch

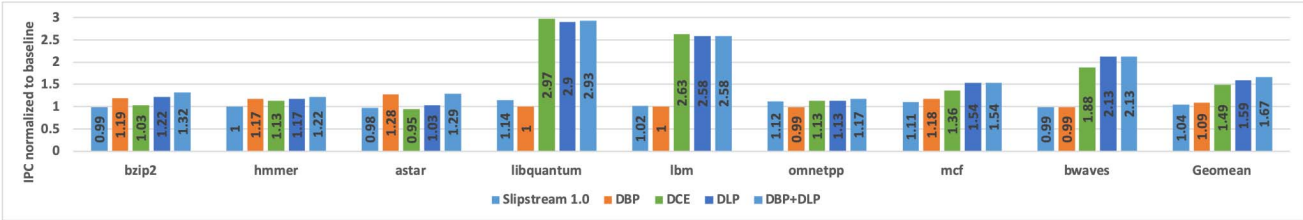


Figure 7: Comparisons among Slipstream 1.0, DCE, and Slipstream 2.0 (DBP, DLP, and DBP+DLP).

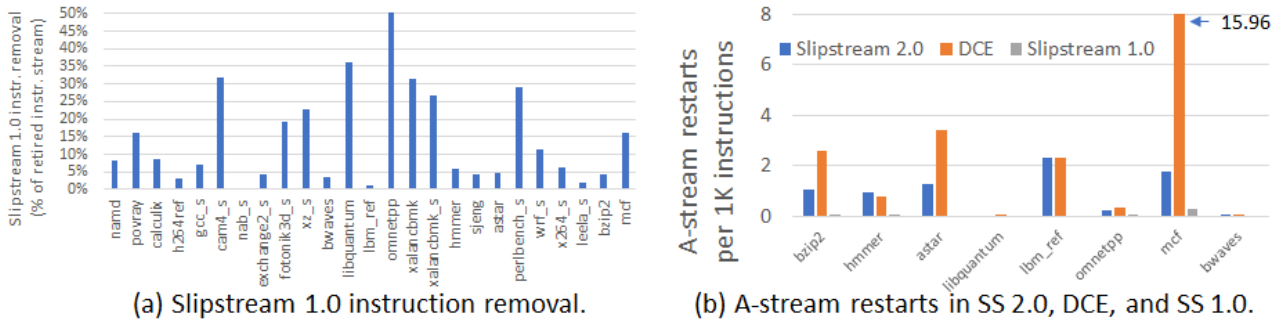


Figure 8: Instruction removal (slipstream 1.0) and A-stream restarts-per-1K-instructions (all).

was mispredicted, exposing the prefetch's latency in the misprediction penalty. DLP elides the dependent branch and its CD region in the A-stream and lets them resolve locally in the R-stream.

Bzip2, astar, and mcf, show 20% to 30% better speedups with DBP+DLP compared to DCE. For applications that do not have many branch mispredictions, such as libquantum, lbm, and omnetpp, DBP+DLP performs within 5% the speedup provided by DCE. Thus, DBP+DLP effectively builds upon DCE and achieves better speedups for control-bound applications as well: DBP eliminates mispredictions of pre-executable branches and DLP extends DCE's latency tolerance to misses feeding mispredicted branches. Overall, DBP+DLP provides a geometric mean speedup of 12% compared to DCE.

C. Microarchitectural Turbo Boost

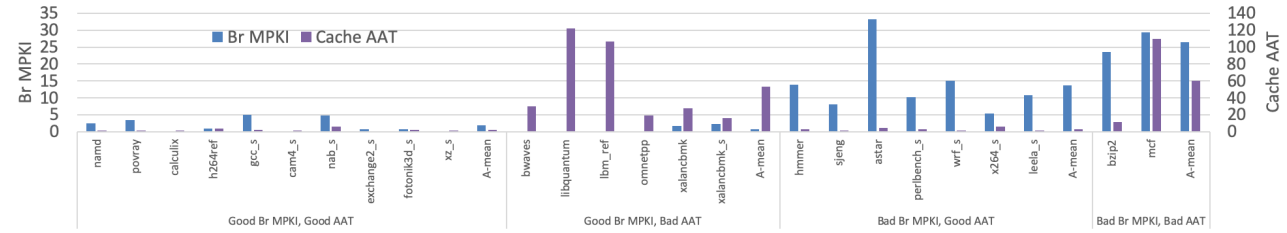
The heuristic used to turn on/off the A-stream (Fig. 10), is based on exceeding MPKI thresholds of DBP-classed branches and DLP-classed loads during each epoch. A-stream utilization is shown in Figure 9b for all 25 benchmarks. 8 benchmarks enable A-stream for 100% of execution time and show speedups with DBP+DLP (Figure 9c). 17 benchmarks disable A-stream for 60% or more of execution time: (i) 10 have low branch and load MPKIs (Figure 9a) and are either high-IPC or bound by true data dependencies. (ii) 3 have delinquent branches or loads only during certain phases and hence had A-stream enabled for 10%-40% of execution time. (iii) 4 (perlbench, sjeng, leela_s, wrf_s) have high MPKI among non-pre-executable branches (ineligible for DBP), hence, the heuristic correctly disables the A-stream for these. Thus, turbo enables Slipstream 2.0 for low-IPC applications that benefit from DBP and DLP.

D. Energy measurements

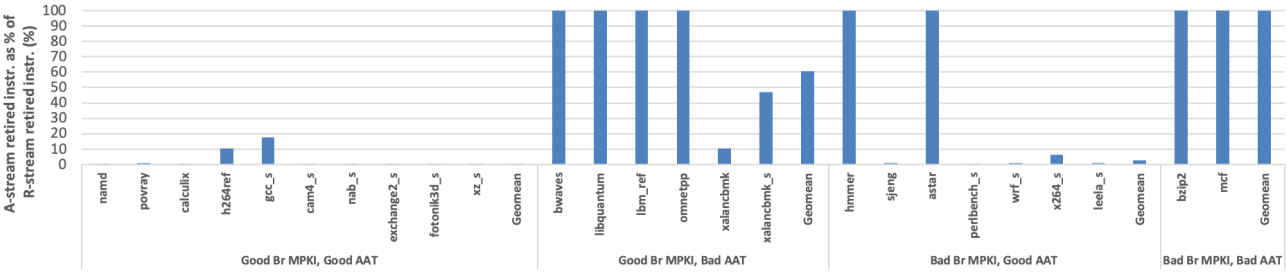
Figure 11 shows energy and energy-delay-product (EDP) of Slipstream 2.0 (two cores) normalized to the baseline (one core). Despite using two cores and additional (albeit quite small) components, the average energy expenditure is 4% less than the baseline with a single core. The main reason is that the reduced execution time leads to lower static energy. There is an increase in dynamic energy as a result of redundant execution. But the benefit from reduced static energy outweighs the increase in dynamic energy significantly. In astar, bzip2, and hmmer, we expend 12% to 24% more energy, but if we factor-in the speedups and measure EDP we see that they do better than the baseline. Memory-bound applications that benefited the most from DLP see a significant drop in energy. Their EDP is much lower than the baseline as well. On average, Slipstream 2.0 reduces total energy by 4% and EDP by 43% compared to the baseline.

V. Summary and Future Work

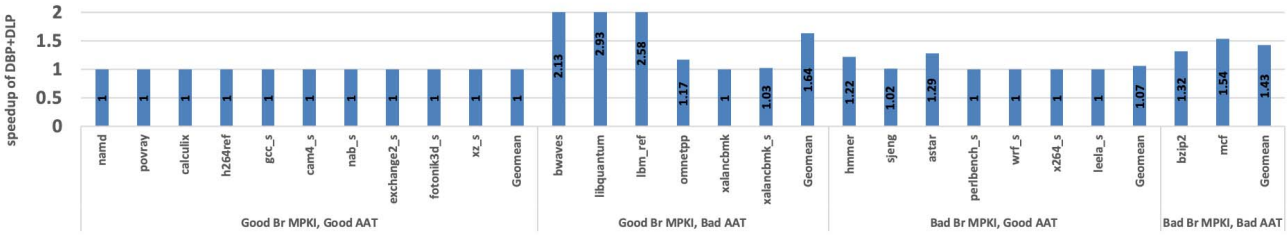
We presented Slipstream 2.0, a new pre-execution microarchitecture that meets four criteria: (i) retains the simpler coordination of a leader-follower microarchitecture as compared to per-dynamic-instance helper threads, (ii) is fully automated with just hardware, (iii) targets both branches and loads, (iv) is effective in exploiting that which is targeted. We reviewed prior pre-execution proposals and showed none of them meet all four criteria. Slipstream 2.0's key innovation in the space of leader-follower architectures



(a) Branch Mispredicts per 1K instructions (MPKI) and Cache Average Access Time (AAT) across applications.



(b) A-stream utilization for slipstream processors 2.0.



(c) Speedup of Slipstream 2.0 DBP+DLP relative to the baseline.

Figure 9: Slipstream 2.0 as microarchitectural turbo boost.

```
// EN: A-stream is enabled
// DBPt: DBP MPKI threshold met in epoch
// DLPt: DLP MPKI threshold met in epoch
if (end of epoch reached) {
   if (!EN && (DBPt || DLPt))
      EN=true;  // enable A-stream
   else if (EN && !(DBPt || DLPt))
      EN=false; // disable A-stream
}
```

Figure 10: Heuristic for enabling/disabling A-stream.

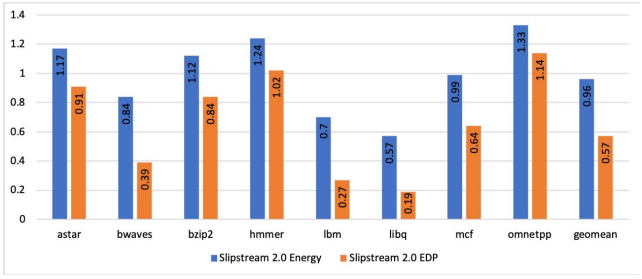


Figure 11: Energy and EDP of Slipstream 2.0 (dual cores) normalized to baseline (single core).

is to remove the *forward control-flow slices* of pre-executable delinquent branches and delinquent loads, from the leading thread.

In addition to being the first pre-execution proposal that meets all four criteria, it includes a simple auto-enable/disable mechanism making it a useful microarchitectural turbo-boost feature and it economically reduces the leading thread by a simple adaptation of conventional branching ("branch-to-reconvergent-point" instead of "branch-to-target").

We compared Slipstream 2.0 to the baseline single core and the only other hardware-only leader-follower prior works in pre-execution: Slipstream (targets branches) and Dual Core Execution (DCE) (targets loads). For SPEC 2006/2017 SimPoints wherein Slipstream 2.0 is auto-enabled, it achieves geomean speedups of 67%, 60%, and 12%, over baseline, Slipstream, and DCE. For SimPoints with primarily delinquent branches, Slipstream 2.0 is 34%, 23%, and 21%, faster than baseline, Slipstream, and DCE. For SimPoints with primarily delinquent loads, Slipstream 2.0 is 84%, 73%, and 9%, faster than baseline, Slipstream,

and DCE. It gives an average reduction of 43% in Energy Delay Product (EDP) and 4% in energy compared to baseline. Finally, it adds only 4.2KB in storage cost for the IR-detector, IR-predictor, and Delay Buffer.

An important byproduct of this work is understanding that only a certain class of delinquent branch can be effectively pre-executed. A delinquent branch is "pre-executable" if it is not in its own forward control-flow slice, *i.e.*, future dynamic instances of the branch are not data-dependent on the outcomes of previous dynamic instances. Moreover, if there are two pre-executable delinquent branches, 'A' and 'B', and 'B' is either control-dependent or control-independent data-dependent (CIDD) on 'A', then 'A' and 'B' cannot both be selected for pre-execution, because 'B' is in the forward control-flow slice of 'A'. Thus, along with the more well-known problem of dependent load misses, non-pre-executable branches remain a performance limiter for Slipstream 2.0 and, we posit, any previous pre-execution microarchitecture. This insight points to much needed future work.

VI. Acknowledgments

This research was supported in part by NSF grant no. CCF-1823517, and grants from Intel. Any opinions, findings and conclusions or recommendations expressed herein are those of the author and do not necessarily reflect the views of the National Science Foundation or Intel.

References

[1] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *Proceedings of the 48th International Symposium on Microarchitecture*, pp. 141–152, December 2015.

[2] A. Seznec, "Tage-sc-l branch predictors again," in *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, June 2016.

[3] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A study of slipstream processors," in *Proceedings of the 33rd International Symposium on Microarchitecture*, pp. 269–280, December 2000.

[4] H. Zhou, "Dual-core execution: building a highly scalable single-thread instruction window," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pp. 231–242, September 2005.

[5] A. Garg and M. C. Huang, "A performance-correctness explicitly-decoupled architecture," in *Proceedings of the 41st International Symposium on Microarchitecture*, pp. 306–317, November 2008.

[6] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi, "Slice-processors: An implementation of operation-based prediction," in *Proceedings of the 15th International Conference on Supercomputing*, pp. 321–334, June 2001.

[7] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads," in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 14–25, June 2001.

[8] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *Proceedings of the 49th International Symposium on Microarchitecture*, pp. 1–12, October 2016.

[9] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pp. 37–48, January 2001.

[10] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th International Symposium on Computer Architecture*, pp. 2–13, June 2001.

[11] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (ssmt)," in *Proceedings of the 26th International Symposium on Computer Architecture*, pp. 186–195, May 1999.

[12] R. S. Chappell, F. Tseng, A. Yoaz, and Y. N. Patt, "Difficult-path branch prediction using subordinate microthreads," in *Proceedings of the 29th International Symposium on Computer Architecture*, pp. 307–317, May 2002.

[13] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault tolerance," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257–268, November 2000.

[14] V. K. Reddy, S. Parthasarathy, and E. Rotenberg, "Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 83–94, October 2006.

[15] R. Parihar and M. C. Huang, "Accelerating decoupled look-ahead via weak dependence removal: A metaheuristic approach," in *Proceedings of the 20th International Symposium on High-Performance Computer Architecture*, pp. 662–677, February 2014.

[16] S. Kondguli and M. Huang, "R3-dla (reduce, reuse, recycle): A more efficient approach to decoupled look-ahead architectures," in *Proceedings of the 25th International Symposium on High-Performance Computer Architecture*, pp. 533–544, February 2019.

[17] 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 *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pp. 129–140, February 2003.

[18] J. D. Collins, D. M. Tullsen, and H. Wang, "Control flow optimization via dynamic reconvergence prediction," in *Proceedings of the 37th International Symposium on Microarchitecture*, pp. 129–140, December 2004.

[19] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "Slipstream memory hierarchies," Tech. Rep. CESR-TR-02-3, Department of Electrical and Computer Engineering, North Carolina State University, February 2002.

[20] 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 *Proceedings of the 42nd International Symposium on Microarchitecture*, pp. 469–480, December 2009.