2025 IEEE International Symposium on High Performance Computer Architecture (HPCA) | 979-8-3315-0647-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/HPCA61900.2025.00015

Delinquent Loop Pre-execution Using Predicated Helper Threads

Anirudh Seshadri and Eric Rotenberg
Department of Electrical and Computer Engineering, North Carolina State University
{aseshad2,ericro}@ncsu.edu

***Abstract*—Branch pre-execution targets delinquent branches that are not predictable by conventional branch predictors. Helper threads attempt to resolve branches ahead of the main thread. Pre-executed branch outcomes are communicated to the main thread's fetch unit via a global branch queue or local branch queues (one per branch PC). Two key challenges are discussed in this paper.**

1) ***Handling a delinquent branch b2 that is control-dependent on another delinquent branch b1.*** **Prior works that include both branches resort to branch prediction of b1 in the helper thread to determine whether or not to pre-execute b2. But b1 is hard-to-predict and the misprediction bottleneck merely shifts from the main thread to the helper thread.**
2) ***Handling a store instruction that both influences a delinquent branch and is control-dependent on it.***

We propose *predicated helper threads (Phelps)* to address these challenges. Phelps constructs a helper thread for each inner loop containing delinquent branches. All delinquent branches, even control-dependent ones (b2), are unconditionally pre-executed in each loop iteration. Per-branch queues are managed in lock-step based on loop iterations, allowing the helper thread to deposit outcomes for both b1 and b2 each iteration and the main thread to consume or ignore b2 outcomes in the correct sequence dictated by b1. The helper thread also retains influential stores for dynamic disambiguation and store-load forwarding. Any such store that is control-dependent on a delinquent branch is predicated on the branch's outcome, which is necessary because the helper thread no longer has control-flow (except for the loop branch). Phelps also features dual decoupled helper threads for outer-inner loop pairs, for effective branch pre-execution when the inner loop has a short and unpredictable trip count.

I. Introduction

Delinquent branches are branches that execute frequently and are also frequently mispredicted by state-of-art branch predictors. A typical example is a branch that tests arbitrary data from a very large data structure such as a graph. A high-performance deep and wide superscalar processor spends tens to hundreds of cycles fetching and executing instructions down the incorrect path of each mispredicted branch. Thus, delinquent branches severely degrade performance and waste significant energy.

If a delinquent branch has a simple control-dependent (CD) region, such as a single or a few arithmetic-logic-unit (ALU) instructions, static or dynamic predicated execution (*e.g.*, [3], [8], [18], [25]) is a good solution. On the other hand, if its CD region is large and/or complex (nested conditional branches, function calls, loads, and stores), predication is either not expressible or not profitable.

One major technique to target delinquent branches with non-trivial CD regions is branch pre-execution [6], [7], [13], [19], [33]–[35], [38], [45], [46], [48]. Branch pre-execution involves learning the backward slice of a delinquent branch, executing dynamic instances of the slice via one or more microarchitectural helper threads ahead of the architectural main thread, and streaming predictions from the helper thread(s) to the main thread's fetch unit via queue(s).

The aim of this work is to highlight two technical challenges that limit the efficacy of branch pre-execution, and propose a new automated, hardware-only, helper thread microarchitecture to address them: (1) delinquent branches that are control-dependent on other delinquent branches, and (2) store instructions that both influence delinquent branches and are control-dependent on them.

Figure 1 shows an example of two delinquent branches b1 and b2. Branch b2 is control-dependent on branch b1. We say that b1 *guards* b2, and that b1 is the *guarding branch* and b2 is the *guarded* branch. Pointers x and y vary arbitrarily each loop iteration. Thus, some (but not all) instances of store s1 conflict with future instances of the load in the backward slice of b1 (the load *y), and the number of iterations separating the conflicting store and load varies. We will use this example to discuss how dependent delinquent branches and stores pose problems for previous branch pre-execution works and how we approach the problems.

```
for (…) {
   // pointers x and y vary each iteration
   if (*y >= …) { // branch b1
      if (…) {    // branch b2
         *x = …   // store s1
         // other statements
      }
   }
}
```

Fig. 1: Example of dependent delinquent branches and stores.

Dependent delinquent branches: Prior works, that include both branches [13], [34], [46], *predict b1 in the helper thread* in order to make a timely decision of whether or not to pre-execute b2. But b1 is hard-to-predict. When the prediction is incorrect, the helper thread must squash and roll back. *Prediction in the helper thread merely shifts the misprediction bottleneck from the main thread to the helper thread.* Our solution is to always pre-execute b2. Thus, our helper thread generates pre-executed outcomes for both b1 and b2 for each

979-8-3315-0647-6/25/$31.00 ©2025 IEEE
DOI 10.1109/HPCA61900.2025.00015

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

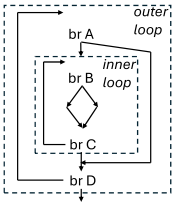
iteration of the loop. The queues for b1 and b2 are managed in lock-step based on loop iterations. Loop-iteration-driven queues is the key mechanism that: (1) allows the helper thread to deposit b1 and b2 outcomes for each iteration of the loop, and (2) allows the main thread to consume some b2 outcomes and ignore others in the correct manner. Although our helper thread pre-executes more instances of b2 than are ultimately needed, it is free from branch prediction and *rollback-free*, hence faster than if it relied on branch prediction of b1.

Dependent stores: Some prior works [7], [38], [48] exclude stores from their helper threads. Slipstream [46] and Decoupled Lookahead [13] feature a continuous leading thread – a pruned version of the main thread – with stores. Including s1 in the leading thread is moot, however: the leading thread is slowed by all branch mispredictions of b1 and b2 [45]. Slipstream 2.0 [45] also includes stores, but not s1: its leading thread skips b1's control-dependent region altogether. Thus, not only does the main thread not benefit from pre-execution of b2, but instances of b1 that depend on prior instances of s1 may have incorrect outcomes. Branch Runahead [34], upon observing the first occurrence of a memory dependence between s1 and the load in b1's slice, assumes the dependence is fixed. It removes the store and load, and directly links the store's producer to the load's consumer. Assuming a fixed memory dependency, rather than keeping the store and load and dynamically disambiguating them in the helper thread, may lead to many incorrect b1 outcomes. Our solution is to include s1 in the helper thread and explicitly predicate it on b1's and b2's outcomes, which is necessary given that our helper thread no longer has control-flow (except for the loop branch).

We call our approach *predicated helper threads (Phelps)*. Phelps constructs a helper thread for each inner loop containing delinquent branches. By targeting loops, we can apply loop-iteration-driven queues, which enable prediction-free/rollback-free pre-execution of nested delinquent branches (b1 and b2). A helper thread also retains influential stores for dynamic disambiguation and store-load forwarding (without affecting architectural state of the main thread), and any such store that is control-dependent on a delinquent branch is predicated.

Figure 2 shows an idiom that occurs in some graph workloads: an inner loop with a short and unpredictable trip-count nested inside a long-running outer loop. Both the outer and inner loops contain delinquent branches. At a minimum, the outer loop contains the inner loop's unpredictable header branch (brA) that determines whether the inner loop will be visited in a given iteration of the outer loop. The inner loop's backward branch (brC) is unpredictable and the inner loop body often contains one or more unpredictable branches (brB).

Targeting just the inner loop would be ineffective because the overhead of starting and stopping a helper thread for each visit to the inner loop is not amortized as it is for a long-running loop. The overhead could be addressed by applying a single helper thread that encompasses both the outer and inner loops, but mispredictions of the inner loop's delinquent loop



br A	inner loop's header branch	delinquent
br B	if-statement	delinquent
br C	inner loop's backward branch	delinquent
br D	outer loop's backward branch	mostly taken

Fig. 2: Nested loop idiom in graph workloads.

branch, brC, become a bottleneck in the single helper thread.

Phelps targets nested loops with dual decoupled helper threads to: (1) incur helper thread start/stop overhead only once for the nested loop as a whole, and (2) tolerate mispredictions of brC. An *outer helper thread* runs the outer loop and queues multiple visits to the inner loop for an *inner helper thread* to process. The inner helper thread executes only one inner loop visit at a time and in program order. While mispredictions of brC are serialized within the inner thread, the outer thread's progress is unaffected by the inner thread. The dual decoupled helper threads achieve high instruction-level parallelism (ILP) and memory-level parallelism (MLP) despite brC, yielding effective branch pre-execution for the main thread.

II. Related Work

Branch pre-execution: Zilles and Sohi characterized backward slices of delinquent branches and loads [49]. They then explored performance potential by *manually* constructing speculative slices and selecting fork points [48]. Slices have no control-flow or stores. They omit *non-delinquent* branches that guard the targeted delinquent branch, which causes extra outcomes to be generated for it. All branch queues must look for specific kill PCs in the main thread (a CAM searched by fetch bundle PCs), to infer when an outcome must be discarded. As with slices and fork points, the kill PCs were manually identified. The issue of nested *delinquent* branches is not discussed. It seems possible to have two concurrent slices (in two thread contexts), and apply the above kill principle to the guarded slice. Our solution is automated and hardware-only, features a novel method (iteration-driven lock-step queues) that avoids kill PC complexity, works with one thread for all nested branches, and handles store s1.

Roth and Sohi used off-line program trace analysis to construct small slices without control-flow or stores that target delinquent loads, branches, or both [38]. In the rename stage, a method called integration allows the main thread to reuse values produced by the helper thread and resolve mispredicted branches early, thus not entirely hiding the misprediction penalty. There is no mention of whether or not nested delinquent branches can be included in Phelps-like fashion. Even if possible, the topic of how to deal with excess outcomes of the guarded branch at the fetch unit was not broached.

Chappell et al. [7] used hardware to characterize all paths leading to a branch, where a path is comprised of multiple basic blocks, and construct specialized slices for those paths for which the targeted branch is hard-to-predict. Per-path

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

specialization can lead to an explosion in the number of slices for a given delinquent branch, especially if it is delinquent no matter the path leading to it. A slice is terminated at a store that conflicted with a load in the slice, to delay forking until after the store.

In Slipstream [35], [46], Decoupled Lookahead (DLA) [13], [19], [33], and Slipstream 2.0 [45], the helper thread (leader) is a pruned version of the main thread (follower) and may include stores. Slipstream prunes the leader by removing highly predictable branches, predictably ineffectual instructions, and their backward slices. DLA uses similar pruning criteria, but by using off-line analysis, may be able to prune more. For predominantly delinquent regions, however, there is little to prune [45]. To address this, Slipstream 2.0 removes the control-dependent regions of delinquent branches, but this causes the pruning of nested delinquent branches and influential stores in these regions.

Branch Runahead [34] constructs one or more chains for a delinquent branch. During backward slicing, a chain is terminated at either a guarding branch, an affector branch (a branch that doesn't guard the delinquent branch, but affects values used by it), or the prior instance of the delinquent branch. If terminated at a guarding or affector branch, chains are constructed for them as well, and so on. Thus, a chain contains no branches besides its terminal branch. Each child chain is tagged with its parent chain (its guard, affector, or self) and parent direction that will trigger it. A bimodal branch predictor is used to predict the parent chain's direction to speculatively trigger its child chains. If incorrect, Branch Runahead must squash all child chains (and their child chains) of the mispredicted parent, and the parent triggers the correct child chains late. A store that conflicts with a load is included in the backward slicing process, but a dependency is assumed to always exist: the store and load are removed, and the store's producer is linked directly to the load's consumer. The assumption may lead to incorrect branch outcomes.

PFM [20] promotes designing RTL for custom branch predictors, data prefetchers, *etc.*, and synthesizing them to a reconfigurable fabric coupled to key pipeline stages of the superscalar processor. The demonstrated custom *astar* branch predictor is, in essence, pre-execution via a "helper thread" implemented as fixed hardware. Impressively, its performance equaled that of perfect branch prediction. The costs are manual per-benchmark design and synthesis of custom branch pre-execution and dedicated hardware of the reconfigurable fabric.

Opportunistic Early Pipeline Re-steering [15] focuses on branches that depend on a single load and simple computation. When the load executes in the backend, it triggers pre-execution of the branch's slice in a dedicated engine in the frontend. Because load-branch pairs are often close in the fetch stream, however, this approach is only timely enough to override the branch predictor for 7% of such dynamic branches. Thus, if pre-execution is too late to override the branch predictor, fetch re-steering is attempted when the branch reaches the last stage of the frontend, reducing but not eliminating the misprediction penalty. To improve prospects, the load's address is predicted in the frontend and its value prefetched from the data cache (if the address prediction is confident). This optimization is shown to predominantly increase late re-steers, not predictor overrides. Triggering pre-execution at the load for each proximal load-branch pair yields a modest average MPKI reduction of 12.7%, which appears to count both predictor overrides and late re-steers. The approach also requires a dedicated execution engine in the frontend.

Load pre-execution: Load pre-execution can be in the form of runahead mode when a cache-missed load reaches the ROB head [28], [29] or helper threads [10], [16], [17], [26], [30], [38], [47], [48]. Note that Phelps does not explicitly identify delinquent loads for load pre-execution. As explained in Section I, the motivation for dual decoupled helper threads for nested loops is amortizing helper thread start/stop latency and tolerating the inner loop's delinquent loop branch. That said, decoupling increases the scope for memory-level parallelism beyond the core's window size. Allowing the outer loop to run independently can increase the concentration of cache misses within the core's window, that are otherwise distant in the dynamic instruction stream, increasing utilization of miss status holding registers (MSHRs).

Load and/or branch scheduling: The Load Slice Core (LSC) [5] endows an in-order core with Decoupled Access/Execute [43] capability. Loads and address generating instructions (AGIs) are steered to the B queue, stores to the B and A queues, and other instructions to the A queue. Steering AGIs requires learning and caching PCs of instructions in backward slices of loads/stores. Iterative Backward Dependency Analysis (IBDA) learns slices iteratively exploiting loops. We borrow IBDA for growing our helper threads. Freeway [21] improves upon LSC by steering load slices that depend on other load slices to a third Y queue. This increases MLP further by unblocking independent load slices in the B queue. Forward Slice Core [22] diverts forward slices of cache-missed loads to a holding queue instead of steering backward slices. All three works attempt to approach MLP of an OOO core with an in-order core.

With ISA support, CRISP [24] uses software to mark the backward slices of critical loads and branches, and the processor's scheduler prioritizes these instructions. This reduces branch misprediction penalties but does not eliminate them.

Value-based branch prediction: Dynamic Data Dependence Tracking [9] propagates dependence bit vectors as instructions are renamed, such that an instruction has a list (in bit vector form) of instructions *currently in the pipeline* that are in its backward slice. The authors suggest multiple applications and specifically explore a value-based overriding branch predictor. A branch's outcomes are recorded for past combinations of input register values to its backward slice. When a particular combination of input register values is observed again, the corresponding outcome is reused. Thus, DDDT-based branch slicing drives a form of computation reuse [11], [14], [44] rather than pre-execution. Reuse may suffer capacity and compulsory misses if there is an explosion in the number of input value combinations. The predictor is

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

elaborate, assuming renaming/DDDT at the instruction fetch stage, a series of circuits, and a redundant register file for obtaining slice input values (if ready) to search the prediction table, necessitating its use as a long-latency overriding predictor of a conventional single-cycle predictor. Accuracy is another concern: input values of the backward slice may not be available due to close proximity of the branch and its slice.

III. Example: *astar*

Figure 3 shows an example of dependent delinquent branches and stores from the SPEC benchmark, *astar*. We refer to this example throughout the paper.

One iteration of the for-loop (line 2) tests attributes of the 8 cells surrounding the cell at *index* (gotten from the input worklist at line 4) in a grid. The figure only shows complete code for the first neighbor, at lines 6–21. The code at lines 6–21 is repeated 7 more times for different *index1* values (other neighbors), shown in abbreviated form at lines 24–30.

This loop features prominently in our highest-weighted *astar* SimPoint [40]. This SimPoint suffers 29 mispredictions-per-kilo-instructions (MPKI) using a 64KB TAGE-SC-L [39] branch predictor, mostly caused by 16 delinquent branches in Figure 3: the pair of branches, **b1** (line 7) and **b2** (line 8), and the seven other identical branch pairs (**b3+b4**, **b5+b6**, ... **b15+b16**) operating on different values for *index1*.

```
 1:  bound2l=0;
 2:  for (i=0; i<bound1l; i++)
 3:  {
 4:    index=bound1p[i];                    branch b1
 5:
 6:    index1=index-yoffset-1;
 7:    if (waymap[index1].fillnum!=fillnum)
 8:      if (maparp[index1]==0)             branch b2
 9:      {
10:        bound2p[bound2l]=index1;
11:        bound2l++;                       store s1
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:    index1=index-yoffset;   [b3,b4,s2]
25:    index1=index-yoffset+1; [b5,b6,s3]
26:    index1=index-1;         [b7,b8,s4]
27:    index1=index+1;         [b9,b10,s5]
28:    index1=index+yoffset-1; [b11,b12,s6]
29:    index1=index+yoffset;   [b13,b14,s7]
30:    index1=index+yoffset+1; [b15,b16,s8]
31:  }  // end for loop
```

Fig. 3: Code fragment from *astar*'s makebound2() function.

The first challenge is that **b2** is *control-dependent* on **b1** (likewise for the other branch pairs). It would not be a challenge if only one or the other were delinquent, but both are. The second challenge stems from the stores **s1** (line 13) through **s8**. These stores to *waymap[index1].fillnum* occasionally influence future load instructions that feed odd-numbered branches **b1**, **b3**, ... **b15**. Namely, there is a loop-carried store-load dependence whenever a load of *waymap[index1].fillnum* uses an *index1* that was visited for the first time in some past loop iteration. Further complicating matters is that **s1** is control-dependent on **b1** and **b2**, and likewise for the seven other stores.

IV. Overview of Phelps

In Section I, we discussed limitations of prior work with respect to (1) nested delinquent branches and (2) stores that both influence them and are control-dependent on them. In this section, we give an overview of how Phelps addresses these challenges.

A. Preliminary Information

Before delving into the overview, we need to explain a few things that are referenced in this section. In this paper, when a helper thread is running, the superscalar core's frontend stages and FIFO structures (Reorder Buffer (ROB), Physical Register File (PRF) free list, Load Queue (LQ), and Store Queue (SQ)) are partitioned for a complexity-effective design and isolated processing of the main thread and helper thread. The Scheduler/Issue Queue (IQ) and execution lanes are flexibly shared. Helper thread stores are committed from its SQ partition to a small private cache (32 doublewords organized in 16 sets, 2-way set-associative). Data evicted from this cache is simply lost. If a helper thread load re-references the same address, it may get stale or up-to-date data, depending on if the main thread's counterpart store already reproduced the data in its L1 cache [36].

B. Key Ideas Behind Phelps

Each inner loop with delinquent branches is identified and a single helper thread is constructed for it. The first instructions (seeds) to be included in the helper thread are all of the loop's delinquent branches and the loop branch. Over multiple iterations of the loop, the seeds' producers are added, their producers are added, and so forth, thereby iteratively building up the seeds' backward slices through register dependencies (IBDA [5]). If a load is added to the helper thread, and a store in the loop is detected to conflict with the load, then the store and its backward slice are also added.

The only control-flow in the helper thread is the loop branch, useful for knowing when to terminate the helper thread. The delinquent branches are converted to predicate-generating instructions, *i.e.*, *predicate producers*, for pushing their pre-executed outcomes to per-branch prediction queues. A key point is that all of the delinquent branches are unconditionally pre-executed, even if they are nested, because their prediction queues operate in lockstep based on loop iterations. That is, each delinquent branch is allocated an entry for each loop iteration so that it may unconditionally deposit pre-executed outcomes for all loop iterations, even if the delinquent branch is supposed to be skipped-over in some loop iterations. The main thread's instruction fetch stage, guided by branch predictions (from the prediction queues for delinquent branches and the core's default branch predictor

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

for non-delinquent branches), dictates which pre-executed outcomes are ultimately consumed, thereby respecting guarding relationships among all branches (whether delinquent or non-delinquent).

Figure 4 shows an example of iteration-based management of the per-branch prediction queues. Each row is a prediction queue for a given delinquent branch in the *astar* example of Figure 3. Only the first four of *astar*'s sixteen delinquent branches are shown for brevity. Each column corresponds to a loop iteration. The helper thread deposits outcomes (0: not-taken, 1: taken) of predicate producers as they retire, in the column pointed to by *tail*. *Tail* is incremented when the helper thread retires an instance of the loop branch, which only occurs after retiring all predicate producers in the current *tail* iteration. *Spec_head* (speculative head pointer) points to the column from which the main thread consumes predictions. Thus, *spec_head* is incremented when the main thread fetches an instance of the loop branch, which only occurs after it has consumed predictions it requires from the current *spec_head* iteration. Notice that some predictions for **b2** and **b4** are shown in parentheses. This is just to highlight the fact that these predictions are not consumed by the main thread (at least not initially: more on this below), because the predictions consumed for their guarding branches **b1** and **b3** are 1 (taken) causing the main thread to not even fetch the corresponding instances of **b2** and **b4**. For example, the main thread is currently consuming predictions from iteration *spec_head*, with the path highlighted by the red arrows. It doesn't consume **b2**'s prediction owing to **b1** being predicted taken. Conversely, it consumes **b4**'s prediction owing to **b3** being predicted not-taken.

Head is incremented, and the corresponding column freed, when the main thread retires an instance of the loop branch. When the main thread recovers from a branch misprediction or load violation (load issued before a conflicting store), *spec_head* is simply rolled back to the mispredicted branch's checkpointed spec_head pointer (in the case of immediate recovery using branch checkpoints) or to *head* (in the case of recovery from the head of the ROB). Importantly, misprediction recoveries in the main thread do not impact the helper thread, and rolling back *spec_head* enables replaying the pre-executed outcomes instead of generating them all over again. This is even true for a branch misprediction in the main thread that stemmed from a helper thread outcome. In fact, this underscores another subtle but important benefit. Though rare, it is possible for the helper thread to generate an incorrect outcome for an instance of **b1** (*e.g.*, due to losing a store's update in the helper thread's tiny data cache before the main thread retires its corresponding store). If that instance's incorrect outcome was "taken", then the main thread initially didn't consume the corresponding **b2**'s prediction. But because all predicate producers are unconditionally executed, fortunately, the prediction for **b2** exists nonetheless and can be revisited by the main thread. After rolling back to the not-taken target of **b1**, the main thread will consume **b2**'s prediction the second time around.

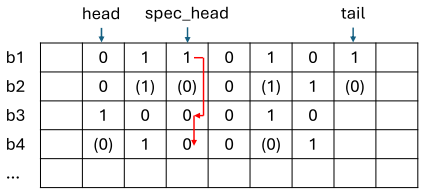


Fig. 4: Prediction queues example.

To support two helper threads (nested loop), there are two sets of {*head*, *spec_head*, *tail*} pointers. Each pointer set pertains to the partition of prediction queues allocated to a given helper thread.

From what has been described thus far, when considering only nested delinquent branches, predication is *implicit* in how the main thread skips consuming some pre-executed outcomes. Explicit predication – linking a predicate producer to a predicate consumer – only comes into play for stores. A store needs to be executed or suppressed depending on the outcomes of the delinquent branches (predicate producers) that guard it. To minimize changes to the core's scheduler, a store may have one predicate source operand, which gets linked to its immediate guarding predicate producer. If the store is nested inside more than one delinquent branch, however, one predicate is insufficient. To compensate, each predicate producer also has one predicate source operand each, which gets linked to *its* immediate guarding predicate producer. Thus, if a store is nested inside multiple delinquent branches, it becomes transitively dependent on all of their predicates. It is important to note that the backward slices of all delinquent branches and included stores are concurrently executed in the helper thread. Only the final instruction of each slice – predicate producer or store – is serialized by its predicate source operand, and only if it is actually guarded. This is illustrated in Figure 5 for *astar*'s **b1**, **b2**, and **s1**.

```
a1: &(boundlp[i])
a5: &wayobj
a7: &waymap
t1: fillnum
t5: yoffset

 1: lw a4,0(a1)      // index=boundlp[i]
 2: subw t0,a4,t5    // index-yoffset
 3: addiw t3,t0,-1   // index1=index-yoffset-1
 4: slli a2,t3,0x2
 5: add t2,a7,a2
 6: lhu s0,0(t2)     // waymap[index1].fillnum
 7: beq s0,t1,1760c  // b1
 8: ld s0,120(a5)    // s0=&maparp
 9: slli s2,t3,0x1
10: add s0,s0,s2
11: lh s0,0(s0)      // maparp[index1]
12: bnez s0,1760c    // b2
13: sh t1,0(t2)      // s1
```

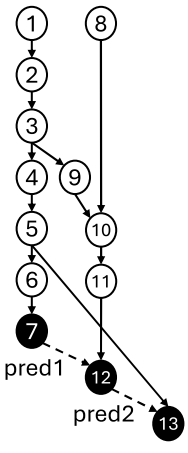


Fig. 5: RISCV assembly and DAG for the backward slices of *astar*'s **b1** (7), **b2** (12), and **s1** (13). The helper thread concurrently executes backward slices of predicate producers (7,12) and stores (13). Only the final instruction of each slice is serialized by its predicate source operand, if guarded by one (12,13).

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

V. Phelps Implementation

A. Epochs

Phelps operates on the basis of fixed instruction intervals called epochs. In this paper, an epoch is 4 million retired instructions of the main thread. The information about delinquent branches and loops gathered in epoch *N* is used to construct new helper threads in epoch *N+1*, which are then available for deployment in subsequent epochs *N+2* and greater.

B. Identifying Loops Containing Delinquent Branches

To identify the most delinquent branches in the current epoch, we use the Delinquent Branch Table (DBT) and Delinquent Branch Table - Max (DBT-Max), shown in Figure 6. The DBT maintains information about conditional branches that mispredicted during the epoch. The DBT is searched by the PC of the branch instruction and each entry consists of a misprediction counter and PC bounds of the inner loop and outer loop that enclose the branch, if applicable. When a conditional branch retires, if it was mispredicted by the core's branch predictor (whether the branch was *actually* predicted by the core's predictor or a prediction queue), it increments its misprediction counter. To train the inner and outer loop PC bounds, the retirement unit keeps track of the PC and target PC of the most recently retired backward branch. When a conditional branch is retired, if its PC is between the backward branch's PC and target PC, the backward branch may replace the inner or outer loop fields of the branch's entry. Which is replaced, if any, depends on comparisons among loops' PC bounds. The two loops whose bounds are closest to the branch are kept, and sorted as inner (tightest loop bounds) and outer (next tightest loop bounds).

PC	misp	inner loop valid	inner loop branch	inner loop target	outer loop valid	outer loop branch	outer loop target
0x11b98	5760	1	0x11bfc	0x11b80	1	0x11c0c	0x11b60
0x11be0	7796	1	0x11bfc	0x11b80	1	0x11c0c	0x11b60

DBT index	misp
1	7796
0	5760

loop branch	loop target	is nested loop	inner loop branch	inner loop target	delinquent branch list	misp
0x11c0c	0x11b60	1	0x11bfc	0x11b80	0x11b98, 0x11be0	13556

Fig. 6: From top to bottom: DBT, DBT-Max, and LT.

DBT-Max ranks the most delinquent branches observed so far. Each entry contains the DBT index of the corresponding delinquent branch, along with its misprediction count for maintaining delinquency ranking. Incrementally updating rankings in the DBT-Max, as mispredicted conditional branches retire and update their counts in the DBT, avoids having to scan the DBT at the end of the epoch to determine the top delinquent branches.

The Loop Table (LT), also shown in Fig. 6, consolidates information about *outermost* loops and the delinquent branches contained within them. It is populated at the end of the epoch, as follows. A pass is made through DBT-Max. During the pass, each delinquent branch in DBT-Max that clears a threshold of 0.5 MPKI (2,000 mispredictions for the 4 million instruction epoch) creates or updates a LT entry for its *outermost* loop branch (gotten from the DBT). The delinquent branch augments the loop's aggregate misprediction count with its misprediction count and adds itself to the loop's delinquent branches list (actually a bit vector representing DBT-Max entries); if the delinquent branch has both an outer and inner loop in its DBT entry, then it adds the nested inner loop information to its outermost loop's LT entry. At the end of this pass, the LT contains information about the outermost loops (either *outer+inner* for nested loops or *inner* only) that contain the most delinquent branches. The DBT and DBT-Max counters are then reset in preparation for the next epoch.

C. Helper Thread Construction

If multiple delinquent loops were identified in the previous epoch, we pick the most delinquent loop among them that doesn't already have a helper thread in the Helper Thread Cache (HTC) (covered in Section V-E). If the chosen loop is a nested loop, two helper threads are constructed at the same time: an *outer thread* and an *inner thread*. If the chosen loop is not a nested loop, just an *inner thread* is constructed. Hereafter, we refer to three helper thread types: *outer-thread*, *inner-thread*, and *inner-thread-only*.

Extracting backward slices requires having both source and destination register specifiers of instructions post-retirement. The ROB has destinations but not sources. Although it is possible to augment the ROB, we opted for a different approach. In any case, the full instructions will eventually be needed, to write the helper threads into the HTC. Therefore, as a preliminary step, all instructions in the loop are collected in the Helper Thread Construction Buffer (HTCB) as they are fetched by the main thread[1]. Only a subset of them will be selected for inclusion in the helper thread as explained next.

Helper thread construction begins with "seed" instructions. For inner-thread-only and inner-thread, the seeds are the inner loop's delinquent branches and backward branch. For outer-thread, the seeds are the outer loop's delinquent branches, its backward branch, and the inner loop's header branch that guards the visit to the inner loop. The inner loop's header branch is included in outer-thread to conditionally queue inner loop visits for inner-thread to process (covered in Section V-F). If a given instance of the header branch is not-taken, outer-thread will queue a corresponding inner loop visit for inner-thread. If taken (branching around the inner loop), outer-thread will not queue an inner loop visit for inner-thread.

[1]Whether fetch is on the right or wrong path, we need to collect all (or as many possible) paths through the loop in any case. This is also why epochs are sufficiently long. The finalized helper thread will be a subset of the loop based on slicing.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

Once the seeds are planted in a helper thread, instructions in their backward slices are added iteratively over multiple loop iterations as in IBDA [5], using the Last Producer Table (LPT). The LPT has as many entries as there are logical integer registers in the ISA. Each entry contains the PC of the most recently retired instruction that is a producer of the corresponding logical register. When an instruction that is already included in the helper thread retires, it reads out the PCs of its producers using its logical source registers as indices into the LPT. Any producers not already included in the helper thread and within the loop's PC bounds are added to the helper thread.

If an instruction references a producer whose PC is outside the loop's PC bounds, the instruction's logical source register is added to the helper thread's live-in register set. A helper thread may have one or two live-in register sets, depending on the helper thread type:

- Inner-thread-only or outer-thread: One live-in register set, for live-ins coming from the main thread.
- Inner-thread: Two live-in register sets, for live-ins coming from the main thread and outer-thread.

To capture store-load dependencies, a 16-entry queue of retired stores is maintained, including the stores' addresses and PCs. Only stores whose PCs are within the loop's PC bounds are captured in this store queue. When a load instruction that is included in the helper thread retires, it looks for a match on its address in this store queue. Upon a match, the matched store is included in the helper thread.

D. Learning Immediate Predicate Producers

As discussed in Section IV-B and depicted in Figure 5, a store is predicated by its immediate predicate producer, if one exists. To handle nesting within multiple branches, each predicate producer must also be predicated by its immediate predicate producer.

We use a novel method to learn immediate predicate producers. It uses the Control-Dependency Finite State Machine (CDFSM) matrix depicted in Figure 7. Each element of the CDFSM matrix is a 2-bit FSM. A row is allocated for each delinquent branch and included store in the loop. A column is allocated for each delinquent branch in the loop.

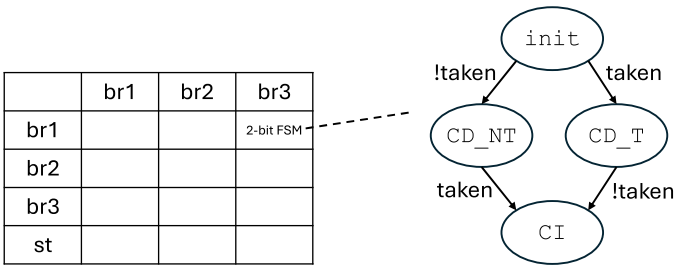


Fig. 7: CDFSM matrix and state transitions of each CDFSM.

The goal is to learn the immediate guarding branch (column) of each branch or store (row). We will use the example in Figure 8a, which shows an abbreviated control-flow graph

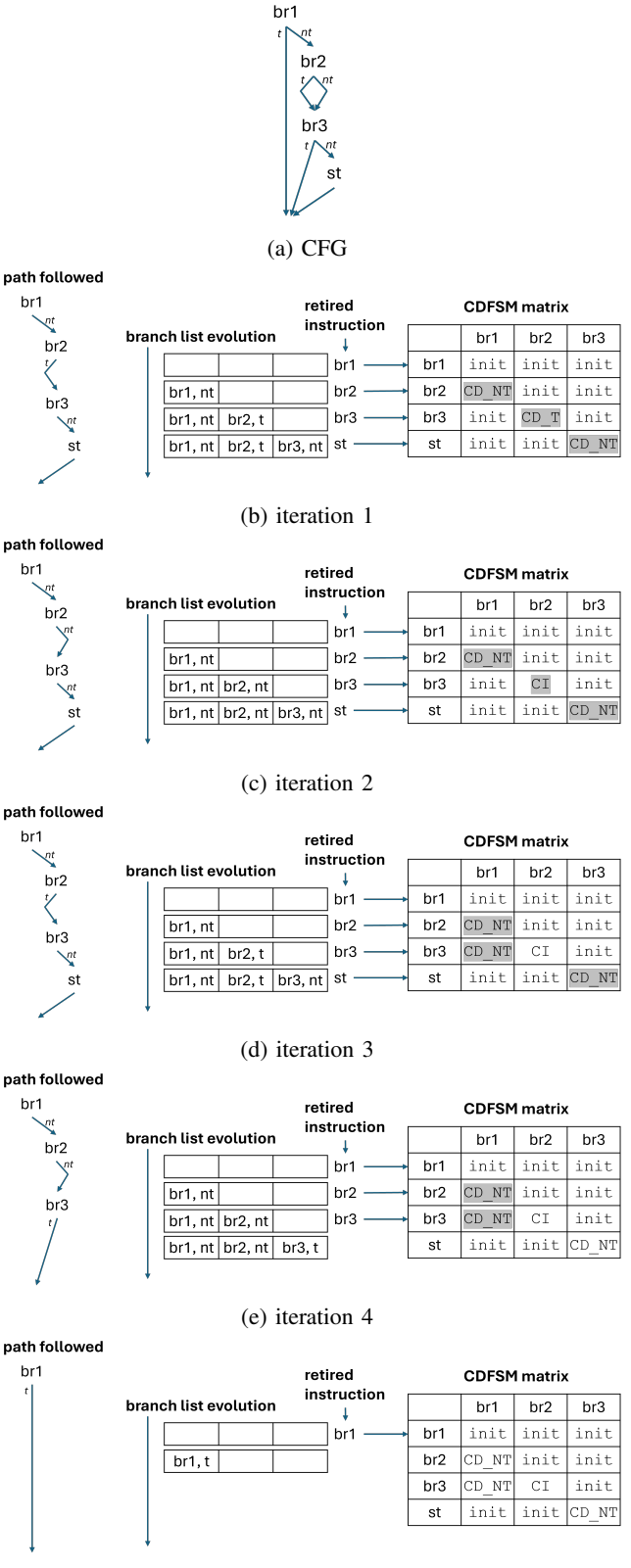


Fig. 8: Example of learning immediate guarding branches. (a) CFG showing three branches and a store inside an example loop (loop branch not shown). (b)-(f) CDFSM training during loop iterations 1 through 5.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

(CFG) for three delinquent branches and a store inside of a loop of interest. Training occurs as instructions in the loop retire and is facilitated by a *branch list*. The branch list is a list of delinquent branches and their directions that have been retired in the current loop iteration. It is cleared for each new iteration, when the loop branch retires. Figures 8b–8f show training over five consecutive iterations of the loop, with the "path followed" indicated on the left for each iteration. In general, more iterations may be needed to observe most or all possible paths, and there are no guarantees.

- Iteration 1 (Figure 8b): When br1 retires, there are no branches prior to it in the branch list (empty branch list); thus, it doesn't train any FSM in its row. When br2 retires, it observes {br1,nt} just prior to it in the branch list; thus, it updates the br1 FSM in its row to CD_NT (highlighted with gray shading for the FSM at row:br2, col:br1), signifying that, based on information so far, br2 appears to be immediately control-dependent on br1 in its not-taken direction. When br3 retires, it observes {br2,t} just prior to it in the branch list; thus, it updates the br2 FSM in its row to CD_T because, based on information so far, br3 appears to be immediately control-dependent on br2 in its taken direction; this is not actually the case, but will be corrected once the alternate path of br2 is observed, below. When st retires, it observes {br3,nt} just prior to it in the branch list; thus, it updates the br3 FSM in its row to CD_NT, *i.e.*, thus far, st deems itself immediately control-dependent on br3 in its not-taken direction.
- Iteration 2 (Figure 8c): This iteration is much the same as the previous one, except that br2 is not-taken. Thus, the only difference is that when br3 retires, it observes {br2,nt} just prior to it in the branch list. Accordingly, it updates the br2 FSM in its row from CD_T to CI. In other words, so far, br3 has always observed br2 just prior to it, and in both directions of br2, so br3 deems itself control-independent (CI) with respect to br2. As we shall see below, in future iterations, br3 must subsequently ignore br2 and train a different FSM, that of a branch prior to br2 in the branch list.
- Iteration 3 (Figure 8d): The path followed in this iteration is the same as iteration 1 and training for br1, br2, and st is the same (no changes in their rows). When br3 retires, it looks past br2 in the branch list because it deems itself control-independent of br2 (CI state). It observes {br1,nt} prior to it and consequently updates the br1 FSM in its row to CD_NT.
- Iterations 4 and 5 (Figures 8e and 8f): These fill out the alternate paths of br3 and br1, respectively, to show that no further changes to the FSMs are possible. In iteration 4, br3 is taken, so st is not retired in this iteration and its row is not trained. In iteration 5, br1 is taken, so br2, br3, and st are not retired in this iteration and their rows are not trained.

The final state of the CDFSM matrix shows that: (1) br1 has no immediate predicate producer (br1 is not guarded by any delinquent branch in the loop); (2) br1 is the immediate predicate producer of both br2 and br3, in the not-taken direction (br1 immediately guards both br2 and br3 and they are on br1's not-taken path); (3) br3 is the immediate predicate producer of st, in the not-taken direction (br3 immediately guards st and it is on br3's not-taken path).

E. Helper Thread Cache

After constructing a helper thread (Section V-C) and learning immediate predicate producers (Section V-D):

1) The instructions selected for inclusion in the helper thread are retrieved from the HTCB.
2) Delinquent branches are converted to predicate producers. Each predicate producer is assigned a unique logical destination predicate register, starting at pred1 and incremented for successive predicate producers (pred0 is reserved as explained below). RISCV [1] note: It can be encoded in bits 11:7, where rd resides for other formats, because the branch's immediate is not needed.
3) Each store and predicate producer is given an extra source operand, for its logical source predicate register. One additional bit encodes the direction of the control dependence (taken/not-taken). The CDFSM matrix indicates which immediate predicate producer the store or predicate producer depends on, and the direction. If there is no immediate predicate producer (not guarded: all FSMs in the row are idle and/or CI), pred0 is assigned to signify unconditional execution. RISCV note: To preserve fixed-length instructions in the HTC and fetch pipeline stages, the new source requires an extra 6 bits due to no available space in the store format. This does not affect the instruction cache or memory hierarchy, however.
4) The final helper thread instructions are written into the HTC.

In this paper, the HTC can hold the helper threads for up to four loops. The HTC is organized as four rows. Each row is dedicated to a loop and holds up to 128 instructions. The loop is tagged with its start PC (the target of the outermost loop branch). A flag indicates whether the loop is a nested loop or non-nested loop. If nested, the row is divided into two halves, and the outer-thread's instructions are packed into the first half and the inner-thread's instructions are packed into the second half. Helper thread instruction fetching is purely sequential until the last instruction (loop branch) is reached, which resets sequencing back to the first instruction. Additional metadata include (1) the locations (instruction offsets in the row) of up to two loop branches (one for inner-thread-only or outer-thread, one for inner-thread), (2) the location of the inner loop's header branch within outer-thread, and (3) up to three live-in register sets (one for inner-thread-only or outer-thread, two for inner-thread).

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

F. Triggering Helper Threads

As the main thread retires instructions, their PCs are compared against the start PCs of the four loops in the HTC. If there is a hit:

1) The pipeline is squashed (no in-flight instructions).
2) Frontend pipeline stages, the LQ/SQ, the ROB, and the PRF free list are partitioned for the main thread and either one helper thread (inner-thread-only) or two helper threads (outer-thread and inner-thread). Table I shows the fractional allocation for the threads' partitions, applied to both frontend pipeline stage width and resources.
3) The fetch stage partition of each helper thread injects annotated move instructions, one per logical register in its live-in register set with respect to the main thread. If the helper thread is inner-thread-only or outer-thread, it immediately starts fetching instructions from its HTC entry after injecting the moves (if inner-thread, it waits). When an annotated move instruction reaches the helper thread's rename stage partition, its source is renamed to the corresponding mapping in the main thread's Rename Map Table (RMT). Its destination is renamed as any destination in the helper thread would be: pop a free register from the helper thread's free list partition and update the helper thread's RMT with the new mapping. Thus, when the move executes, it copies a live-in value from the main thread to the helper thread.
4) Instruction fetching in the main thread only resumes when the last of the annotated moves retires. Though not optimal, this is a simple way to ensure that each helper thread obtained its live-in values before the main thread frees the physical registers from which the values were obtained.

active threads	MT's partition	OT's partition	IT's partition
MT + ITO	$\frac{1}{2}$	n/a	$\frac{1}{2}$
MT + OT + IT	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{3}{8}$

TABLE I: Fractional allocation of frontend width and resources, among main thread (MT), inner-thread-only (ITO), outer-thread (OT), and inner-thread (IT).

For the case of two helper threads, inner-thread is idle until outer-thread queues a visit to the inner loop. Visits are queued in the Visit Queue depicted in Figure 9. When outer-thread retires a not-taken instance of the inner loop's header branch, it allocates a new Visit Queue entry at the tail and writes live-in values (for the inner-thread's second live-in register set supplied by outer-thread) for that visit at slots in the tail entry. When inner-thread determines that its current inner loop visit has fully iterated – *i.e.*, when its loop branch resolves as not-taken – it dequeues the next visit from the head entry. Inner-thread injects move instructions to copy live-in values from the Visit Queue. Each move's source is renamed to the correct slot in the head entry of the Visit Queue, and its destination is renamed to a free physical register obtained from inner-thread's free list partition. Since outer-thread places values in the Visit Queue, it need not wait for inner-thread's moves to read the values.

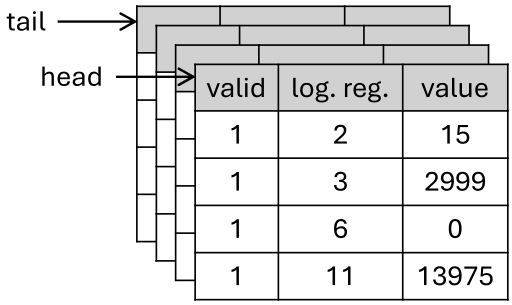


Fig. 9: Visit Queue for outer-thread to queue inner loop visits and their live-in values for processing by inner-thread.

G. Terminating Helper Threads

Pre-execution is terminated when the inner-thread-only or outer-thread loop branch resolves as not-taken, exiting the region of interest. Because non-delinquent loop side-exit branches are not included in the helper threads, the main thread terminates pre-execution when it retires an instruction whose PC is outside the PC bounds of the loop being pre-executed. An exception recovery in the main thread also terminates the helper thread. After terminating the helper threads, the pipeline is squashed and all resources are returned to the main thread.

H. Core Modifications

Modifications to the core include:

- Two extra RMTs to support up to two active helper threads.
- Logical predicate source/destination operands are renamed to physical predicate registers. Thus, we add a predicate physical register file (pred-PRF), predicate physical register free list (pred-FL), and two predicate rename map tables (pred-RMT); when two helper threads are running, pred-FL is partitioned 1/2 for outer-thread and 1/2 for inner-thread. Each predicate register is only 2 bits wide: the most-significant bit (msb) indicates whether the predicate producer was predicated-true (enabled) or predicated-false (suppressed) by *its* predicate producer, and the least-significant bit (lsb) indicates the taken/not-taken outcome of the predicate producer. The consumer of a predicate register is predicated-true if: ((msb == 1) && (lsb == enabling_direction_of_consumer)). Recall, the enabling direction of the consumer is encoded as an extra bit in its predicate source operand (Section V-E).
- The scheduler accommodates an extra source tag in its wakeup CAM, for the extra source operand for stores and predicate producers. (Whether or not this requires more logic, depends on the scheduler style: matrix vs. tag CAM.)
- The core supports horizontal partitioning of frontend pipeline stages and resource partitions. We contend that this should be the case, or else a dedicated frontend and resources, for any pre-execution microarchitecture.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

Component	Section	Parameters	Cost
Components for Helper Thread Construction			
Delinq. Branch Table (DBT)	V-B	256 entries, fully-assoc.	5,280 B
DBT-Max	V-B	32 entries, fully-assoc.	84 B
Loop Table (LT)	V-B	8 entries, fully-assoc.	170 B
Helper Thread Construction Buffer (HTCB)	V-C	256 inst., 4B/inst. metadata	1,024 B 62 B
Last Producer Table (LPT)	V-C	32 entries, 30 bits/entry	120 B
queue to detect needed stores	V-C	16 entries, 94 bits/entry	188 B
CDFSM matrix, branch list, PC-to-row conversion table	V-D V-D implied	32 rows x 16 col. x 2 bits 16 entries, 5 bits/entry 32 entries, 35 bits/entry	128 B 10 B 140 B
Components for Helper Thread Execution			
Helper Thread Cache (HTC)	V-E	4 x 128 inst x 38 bits/inst 4 x 180 bits metadata	2,432 B 90 B
Visit Queue	V-F	16 visits, 4 live-ins/visit, 70 bits/live-in	560 B
Prediction Queues	IV-B	16 queues, 32 iterations 16 PC tags	64 B 60 B
speculative D$ for HT stores	IV-A	16 sets, 2 ways, 8B block metadata	256 B 236 B
pred-PRF, pred-FL, 2 pred-RMTs	V-H V-H V-H	128 reg., 2 bits/reg. 97 entries, 7 bits/entry 2x 31 entries, 7 bits/entry	32 B 85 B 54 B
Total Cost			**10.82 KB**

TABLE II: New components.

I. Summary of New Components

Table II summarizes the new components for Phelps, their parameters used in this paper, and costs.

J. Conditions that Render Loops Ineligible

Three conditions render a delinquent loop ineligible for Phelps. First, if the number of instructions in the helper thread is more than 75% of the number of instructions in the loop, the loop is ineligible for pre-execution. It may not be profitable and may even be detrimental, especially considering half the pipeline's resources are taken away from the main thread. Second, due to high overheads for starting and stopping helper threads (pipeline squashes to reconfigure before and after, main thread stalling until moves retire), a loop is ineligible if it does not iterate enough per visit. Third, a nested loop is ineligible if outer-thread is data-dependent on inner-thread. This scenario is detected when an instruction added to outer-thread references a PC in the LPT that is within the PC bounds of the inner loop.

A loop that is otherwise eligible but exceeds parameter limits (*e.g.*, more live-in registers than can be encoded in the HTCB/HTC metadata) becomes ineligible.

K. Omitting More Complex Scenarios

In our research, we explored and devised methods to support more complex scenarios in Phelps. We omitted these features in this paper because they add significant complexity with no more profitable results. We note, however, that these features are possible.

- We explored including non-delinquent unbiased branches in the helper thread if they guard other instructions added to the helper thread. To implement this, the CDFSM matrix includes rows for all instructions in the loop, and after growing the helper thread for awhile, more branches are added as seeds: those that were not among the original seeds, are unbiased, and guard any instruction in the helper thread thus far. These branches are left as control-flow in the helper thread if predictable with a bimodal predictor, or converted to predicate producers otherwise. Either way, they must be predicate consumers.
- We explored the problem of *alternate producers*. An instruction in the helper thread, that is control-independent of a prior branch, may depend on different producers of its source register depending on the direction of the branch. We detected this by recording the last producer PC alongside each source of each instruction added to the helper thread, and then noting a different PC in the LPT in a future iteration. Both alternate producers are added to the helper thread. If their guarding branches were converted from control-flow to predicate producers, the alternate producers are made predicate consumers; whenever predicated-false, they convert to a move of the previous version of their logical destination register.
- We explored more complex guarding scenarios, such as those created by if statements with OR expressions. This requires multiple predicate source operands, the evaluations of which are ORed together. The scenario is detectable as multiple CD states in a row of the CDFSM matrix.

VI. Methodology

For our experiments, we use an in-house, RISC-V, execution-driven, cycle-level, execute-at-execute simulator to model the superscalar core and the additional components required for modeling Phelps. The superscalar core and memory hierarchy are shown in Table III. Of note is that state-of-the-art high-performance cores have scaled to large windows and superscalar widths. For example, reverse engineering by AnandTech [12] of Apple's A14 chip inferred a ROB size of around 630 instructions and LQ size of 148 loads. Accordingly, the principal superscalar core used in our experiments has a ROB size of 632 and LQ size of 144 (to be divisible by 8 for partitioning among the main thread and helper thread(s)).

branch predictor	64KB TAGE-SC-L
pipeline depth	11 stages (fetch to retire)
fetch/retire width	8 instr./cycle
issue/execute width	8 instr./cycle
execution lanes	4 simple ALU, 2 load/store, 2 FP/complex ALU
ROB/PRF/LQ/SQ/IQ	632/696/144/144/128
L1I cache	32KB, 8-way
L1D cache	48KB, 12-way, 3 cycles (1 agen, 2 hit)
L1D prefetcher	IPCP (16.7 KB) [31], [32]
L2 cache	1.25 MB, 20-way, 15 cycles
L3 cache	3 MB, 12-way, 40 cycles
L2/L3 prefetcher	VLDP (5.5 KB) [41], [42]
DRAM	100 cycles

TABLE III: Superscalar core and memory hierarchy conf.

We perform our experiments on the SPEC 2006 integer benchmark *473.astar* with reference input *rivers* (473.as-

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

tar_rivers), the GAP graph benchmark suite [4], and the SPEC 2017 integer benchmarks. For all GAP benchmarks, we use the roadNet-CA [23] graph as the input dataset. We use the SimPoints methodology [40] to obtain up to 5 representative regions of 100 million instructions each and compute the weighted harmonic mean of IPCs over a benchmark's SimPoints to obtain the overall IPC of the benchmark.

Branch Runahead: We also implemented the core-only version of Branch Runahead [34]. As with Phelps' MT+ITO configuration (Table I), the core's frontend pipeline stages, PRF free list, and LQ are partitioned 50/50 with the main thread executing in one partition and all chains executing in the other (the main thread has the whole ROB and SQ to itself; more below). While Branch Runahead respects program order between parent chains that trigger child chains based on branch dependencies (guard-guarded and affector-affectee relantionships), it loses program order among independent chain groups. Referring back to Figure 3, if stores are ignored (hence, only guard-guarded relationships), *astar*'s makebound2() function has 8 independent chain groups: {b1,b2} is a chain group, {b3,b4} is another chain group, and so on. Figure 10a shows example dynamic sequences of these chain groups. Order is respected within a chain group but not among chain groups. This can be beneficial (exploit control independence, when there is chain group level parallelism) but is a major source of implementation complexity:

- The core's existing physical register management mechanism cannot be used by chains, because committing and freeing of physical registers is driven by the ROB which lists instructions in total program order. Instead, it is necessary to implement a second physical register management mechanism in the chain partition: usage counters [2], [27]. Not only does this mean implementing two register management schemes, but reference counting is challenging [37].
- Independence among chain groups is exploited by selectively rolling back only the affected chain group, which is complex to implement in our experience. Selective rollbacks occur in two cases: (1) the main thread consumed an incorrect prediction from a chain, or (2) a parent chain speculatively triggered an incorrect child chain. The complexity is two-fold: first, selective rollback of only the affected chain group; second, the need to rollback to the top-level (self-dependent) chain in a chain group (instance A′ in Fig. 10b) even if the flaw stemmed from a descendant of the top-level branch (instance B′) – please see Figure 10b.

Figure 11 compares the performance of Branch Runahead (BR) with non-speculative (BR-non-spec) and speculative (BR-spec) triggering, full-featured Phelps, and Phelps with various features omitted, on just the top-weighted *astar* SimPoint (in which makebound2() is prominent). We excluded stores from BR to help it: its approach of replacing def-store-load-use with a fixed def-use would introduce affector-affectee relationships between chain groups {b1,b2}, {b3,b4},

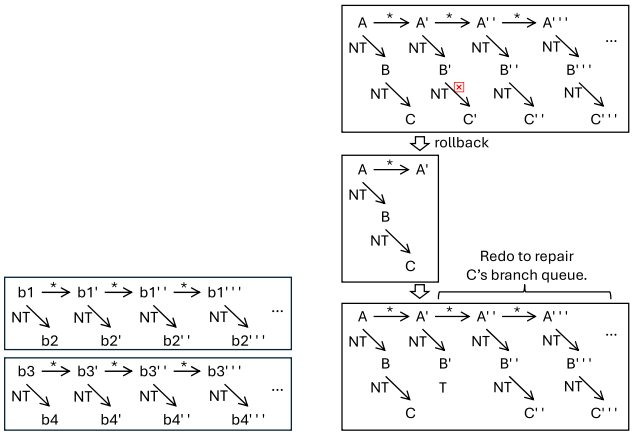


Fig. 10: Branch Runahead: (a) First two of eight chain groups in *astar*. (b) Rollback within an example chain group. B′ speculatively and incorrectly triggered C′. To repair Branch Runahead's branch queue of C, we must rollback to top-level branch A′, to retrigger A″, A‴, and their descendants.

etc.; this would merge them into one chain group, serialize their triggering, and rely even more on inaccurate speculative triggering; worse, fixed def-use would also lead to incorrect predictions. BR-spec's inaccurate triggering of nested chains b2, b4, *etc.*, and resulting rollbacks of their chain groups, is tolerated to some extent by having chain group level parallelism of 8, yielding a good speedup of 29%. Full-featured Phelps (Phelps:b1→b2→s1) achieves higher speedup (47%) with a simpler pre-execution paradigm: a single, self-contained, rollback-free (except for load violations) helper thread that is wholly free of branch prediction's limitations.

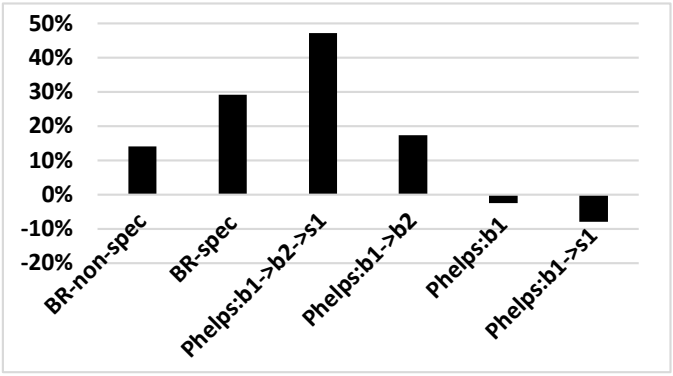


Fig. 11: Phelps and Branch Runahead for *astar*'s top SimPoint.

The last three bars isolate the impact of (1) nested delinquent branches and (2) stores that both affect delinquent branches and are guarded by them. Phelps:b1→b2 signifies that the helper thread pre-executes both b1 and guarded branch b2, but it does not include guarded store s1 (same applies for the seven other groups). Whereas full-featured Phelps reduces MPKI of this SimPoint from 29.5 to 2.68, Phelps:b1→b2 reduces MPKI to only 13.4. Phelps:b1 signifies that the helper thread pre-executes b1, does not pre-execute guarded branch

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

b2, and does not include guarded store s1. It only reduces MPKI to 22.9, which is not enough to offset taking away half the core's resources from the main thread. Phelps:b1→s1 signifies that the helper thread pre-executes b1 but not guarded branch b2, and includes guarded store s1. It reduces MPKI to 24.5; s1 is supposed to be additionally guarded by b2, and b2's omission causes unsuppressed s1 instances that worsen b1's accuracy.

VII. Results

Figure 12a shows the speedups of perfect branch prediction (perfBP), Phelps, and Branch Runahead with speculative triggering (BR). A fourth configuration, BR-12w, features a 12-wide core, in which the main thread gets the same frontend width and resources as the baseline (8-wide frontend, ROB/PRF/LQ/SQ = 632/696/144/144) and BR chains get the same frontend width and resources as the BR configuration (4-wide frontend, PRF/LQ = 316/72); the IQ and 12 execution lanes are still flexibly shared, although there are 4 more lanes.

Phelps yields good speedups on *bc* (63%), *bfs* (64%), and *astar* (15%). *Astar*'s speedup considering all SimPoints is much less than that of the top-weighted SimPoint alone, for two reasons. First, the other SimPoints are not nearly as delinquent. Second, their delinquent loops are not sufficiently long-running per visit to amortize helper thread start/stop overheads, hence they are ineligible (Sec. V-J).

Figure 12b examines the importance of stores. Predicated stores are critical for Phelps to achieve its potential on *bc* and *astar*. They feature stores that both influence and are control-dependent on delinquent branches. *Bfs* does as well, but the store-load distance in the inner-thread is long enough in most cases that the main thread retires its corresponding store before the inner-thread references the data. This isn't always the case: Figure 13a shows that *bfs* accuracy degrades a bit without helper thread stores. The lower accuracy is outweighed by more timeliness owing to fewer helper thread instructions.

BR exhibits mostly slowdowns, except for *astar*. This is due to (1) the main thread getting only half frontend width, LQ, and PRF for the full run, and (2) inaccurate speculative triggering combined with not enough chain group level parallelism, yielding many untimely predictions. BR-12w turns things around with mostly speedups.

As shown in Figure 13a, Phelps achieves significant reductions in MPKI of 72%-91% on four of six benchmarks shown. Similar to other branch pre-execution techniques, Phelps achieves speedups at the cost of partially redundant execution. The overhead, in terms of the number of helper thread instructions retired, is shown in Figure 13b. Phelps incurs a mean overhead of 34.7 million helper thread instructions for 100 million instructions retired in the main thread. Fig. 13c shows IPCs for GAP and astar *without Phelps* (main thread only) when the core is partitioned and not partitioned. Taking away half the resources from the main thread yields a slowdown of 4.1% (*pr*) to 12.8% (*bc*). For SPEC 2017 benchmarks, slowdowns range from 2% (*perlbench*) to 31% (*exchange2*).

The SPEC benchmarks are sorted in Figure 12a from highest to lowest speedup with perfBP. Phelps rarely or never activates helper threads in SPEC 2017. To understand why, Fig. 14 breaks down mispredictions (MPKI) into those that are eliminated by Phelps ("eliminated misp."/white segment) vs. not eliminated (color segments), and isolates the reasons for the latter. This data is for the top-weighted SimPoint of each benchmark.

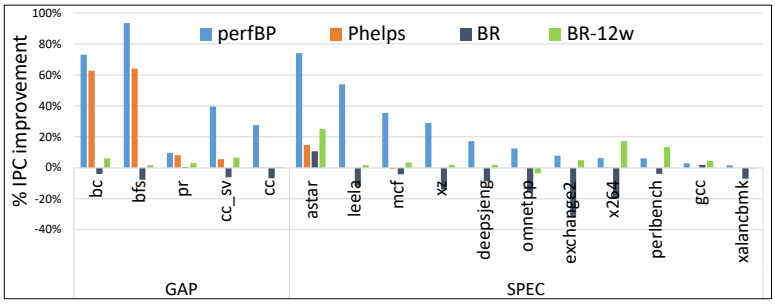
Phelps eliminates most mispredictions in *bc*, *bfs*, *pr*, *cc*, and *astar*. Residual mispredictions are due to the first and second training stages: measuring delinquency ("gathering delinquency"/blue) and constructing a helper thread ("del. but ht being const."/light blue). *Bfs*, *pr*, and *cc* also have a sliver of mispredictions from non-delinquent branches ("not delinquent"/orange).

The purple sliver in *cc_sv* indicates that more than one delinquent loop is detected in the same epoch. Only one is chosen for helper thread construction in the next epoch. Mispredictions from non-chosen loops are classed as "del. but ht not const."/purple until they are selected for helper thread construction. Regardless, the helper threads end up being ineligible for pre-execution because they are too big ("del. but ht too big"/red).

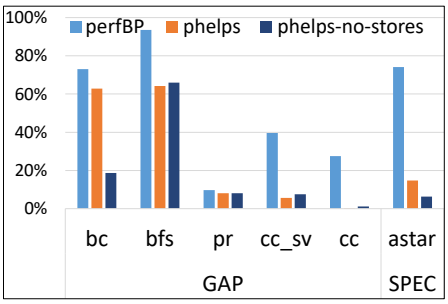
Except for *mcf*, SPEC2017 benchmarks are simply not as delinquent as GAP and *astar*. While *mcf*'s branches are delinquent, most mispredictions are from branches deemed to not be within a loop ("del. but not in loop"/dark green). This may happen when the loop contains a function call and the delinquent branch is within the non-inlined function. In this scenario, the branch's PC is not within the loop's contiguous bounds. For *leela*, *deepsjeng*, *exchange2*, and *xalanc*, the "not delinquent"/orange segment is significant, indicating that much of the MPKI contribution is from branches that are not individually delinquent. The few branches that clear the delinquency threshold in *leela*, *deepsjeng*, and *omnetpp*, end up constructing a helper thread that is too big. Half of mispredictions in *xz* come from non-delinquent branches. The other half come from delinquent branches whose loops don't iterate enough to be eligible for pre-execution ("del. but ot/ito not iterating enough"/light green). A useful helper thread is constructed for *x264* but it is not limited by branch prediction. *Gcc* has too many static branches, causing frequent evictions in the DBT. Thus, most mispredictions come from branches that are still gathering delinquency information (dark blue) and the rest come from non-delinquent branches (orange).

Fig. 15a varies the core's ROB size (with commensurate sizing of PRF/LQ/SQ/IQ). *Bc* and *bfs* show even higher speedups for ROB size 1024, which the baseline is unable to utilize due to frequent squashes. As pipeline depth increases (11, 15, 19), speedups increase for *astar* (15%, 22%, 27%), *bfs* (64%, 70%, 74%), and *bc* (63%, 71%, 79%). Fig. 15b shows speedups for *bfs* with different inputs. SimPoints for web-google capture more of the reinitialization phase between bfs passes. While this phase is delinquent, its helper thread is too big to be eligible.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

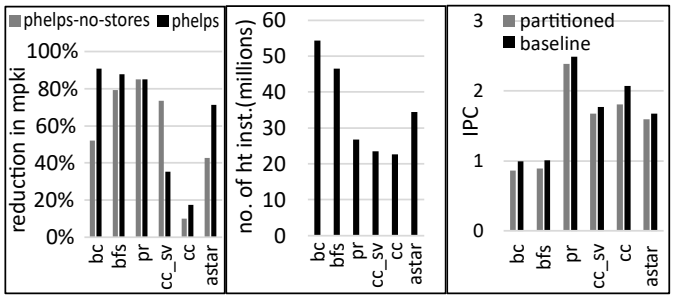


(a) Speedup of perfect branch prediction, Phelps, and Branch Runahead.



(b) Speedup of Phelps with/without stores.

Fig. 12



(a) (b) (c)

Fig. 13: (a) MPKI reduction. (b) Retired helper thread instructions. (c) Isolating the impact of partitioning on main thread.

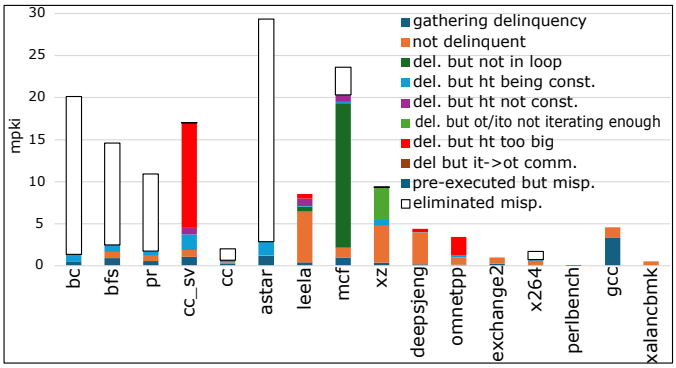
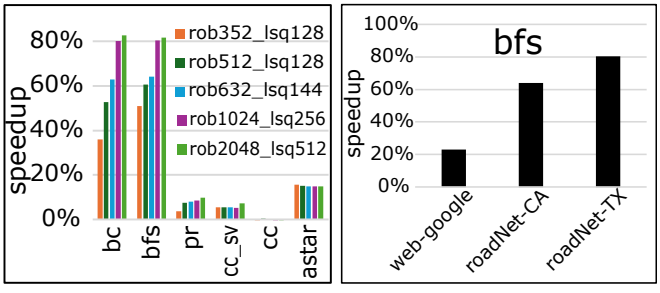


Fig. 14: Characterizing mispredictions (top SimPoints).



(a) (b)

Fig. 15: (a) Sensitivity to window size. (b) Speedups on different inputs for *bfs*.

VIII. Summary

Branch pre-execution is a well-explored technique to handle delinquent branches. However, prior works in branch pre-execution have not adequately addressed the problems of (1) a delinquent branch b2 that is control-dependent on another delinquent branch b1, and (2) a store that both influences a delinquent branch and is control-dependent on it. With respect to (1), prior works either do not discuss dependent branches, pre-execute only the guarding branch b1, or predict the guarding branch b1 which shifts the misprediction botteneck to the helper thread. Phelps' loop-based helper thread unconditionally pre-executes both b1 and b2 each loop iteration, and the main thread consumes or ignores b2 outcomes in the correct sequence based on b1 outcomes. Influential stores are included in the helper thread; if a store is control-dependent on a delinquent branch, it is predicated on that branch's outcome since control-flow has been removed.

A third problem is an inner loop with a short and unpredictable trip count. Helper thread start/stop overhead for each visit to the short loop is not amortized as it is for long-running loops. If the inner loop is nested inside of a control/data independent long-running outer-loop (as in some graph workloads), Phelps applies dual decoupled helper threads for the nested loop. This approach incurs helper thread start/stop overhead only once for the nested loop as a whole and achieves effective branch pre-execution despite the inner loop's delinquent loop branch.

Acknowledgments

This project was funded by the NSF/Intel Partnership on Foundational Microarchitecture Research (FoMR) (NSF grant no. CCF-1823517 and matching Intel grant). Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation or Intel Corporation.

The authors thank the anonymous reviewers for their feedback.

References

[1] The riscv instruction set manual volume i: Unprivileged architecture. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/tag/20240411

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

[2] H. Akkary, R. Rajwar, and S. Srinivasan, "Checkpoint processing and recovery: towards scalable large instruction window processors," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 423–434.

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

[4] S. Beamer, K. Asanović, and D. Patterson, "The GAP benchmark suite," 2015.

[5] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The load slice core microarchitecture," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 272–284.

[6] 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*, May 1999, pp. 186–195.

[7] 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*, May 2002, pp. 307–317.

[8] A. Chauhan, J. Gaur, Z. Sperber, F. Sala, L. Rappoport, A. Yoaz, and S. Subramoney, "Auto-predication of critical branches," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 92–104.

[9] L. Chen, S. Dropsho, and D. Albonesi, "Dynamic data dependence tracking and its application to branch prediction," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 65–76.

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

[11] D. Conners and W. Hwu, "Compiler-directed dynamic computation reuse: rationale and initial results," in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 158–169.

[12] A. Frumusanu, "Apple announces the apple silicon m1: Ditching x86 - what to expect, based on a14," https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2, Nov. 2020.

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

[14] A. Gonzalez, J. Tubella, and C. Molina, "Trace-level reuse," in *Proceedings of the 1999 International Conference on Parallel Processing*, 1999, pp. 30–37.

[15] S. Gupta, N. Soundararajan, R. Natarajan, and S. Subramoney, "Opportunistic early pipeline re-steering for data-dependent branches," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 305–316. [Online]. Available: https://doi.org/10.1145/3410463.3414628

[16] M. Hashemi, O. Mutlu, and Y. N. Patt, "Continuous runahead: Transparent hardware acceleration for memory intensive workloads," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[17] M. Hashemi and Y. N. Patt, "Filtered runahead execution with a runahead buffer," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 358–369.

[18] A. Klauser, T. Austin, D. Grunwald, and B. Calder, "Dynamic hammock predication for non-predicated instruction set architectures," in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192)*, 1998, pp. 278–285.

[19] 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*, February 2019, pp. 533–544.

[20] C. Kumar, A. Seshadri, A. Chaudhary, S. Bhawalkar, R. Singh, and E. Rotenberg, "Post-fabrication microarchitecture," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1270–1281. [Online]. Available: https://doi.org/10.1145/3466752.3480119

[21] R. Kumar, M. Alipour, and D. Black-Schaffer, "Freeway: Maximizing mlp for slice-out-of-order execution," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 558–569.

[22] K. Lakshminarasimhan, A. Naithani, J. Feliu, and L. Eeckhout, "The forward slice core microarchitecture," in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 361–372. [Online]. Available: https://doi.org/10.1145/3410463.3414629

[23] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[24] H. Litz, G. Ayers, and P. Ranganathan, "Crisp: critical slice prefetching," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 300–313. [Online]. Available: https://doi.org/10.1145/3503222.3507745

[25] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu, "A comparison of full and partial predicated execution support for ilp processors," *SIGARCH Comput. Archit. News*, vol. 23, no. 2, p. 138–150, may 1995. [Online]. Available: https://doi.org/10.1145/225830.225965

[26] 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*, ser. ICS '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 321–334. [Online]. Available: https://doi.org/10.1145/377792.377856

[27] M. Moudgill, K. Pingali, and S. Vassiliadis, "Register renaming and dynamic speculation: an alternative approach," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, ser. MICRO 26. Washington, DC, USA: IEEE Computer Society Press, 1993, p. 202–213.

[28] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.*, 2003, pp. 129–140.

[29] O. Mutlu, H. Kim, and Y. Patt, "Techniques for efficient processing in runahead execution engines," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 370–381.

[30] A. Naithani, J. Feliu, A. Adileh, and L. Eeckhout, "Precise runahead execution," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 397–410.

[31] S. Pakalapati and B. Panda. Bouquet of instruction pointers: Instruction pointer classifier-based hardware prefetching. [Online]. Available: https://dpc3.compas.cs.stonybrook.edu/pdfs/Bouquet.pdf

[32] S. Pakalapati and B. Panda, "Bouquet of instruction pointers: Instruction pointer classifier-based spatial hardware prefetching," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 118–131.

[33] 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*, February 2014, pp. 662–677.

[34] S. Pruett and Y. Patt, "Branch runahead: An alternative to branch prediction for impossible to predict branches," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 804–815. [Online]. Available: https://doi.org/10.1145/3466752.3480053

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

[36] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "Slipstream memory hierarchies," North Carolina State University, Tech. Rep., 2002.

[37] A. Roth, "Physical register reference counting," *IEEE Comput. Archit. Lett.*, vol. 7, no. 1, p. 9–12, jan 2008. [Online]. Available: https://doi.org/10.1109/L-CA.2007.15

[38] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceedings of the 7th Annual IEEE International Symposium on High-Performance Computer Architecture*, ser. HPCA '01, 2001, pp. 37–48.

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.

[39] 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.
[40] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, p. 45–57, oct 2002. [Online]. Available: https://doi.org/10.1145/635508.605403
[41] M. Shevgoor, S. Koladiya, R. Balasubramonian, and Z. Chishti. Efficiently prefetching complex address patterns. [Online]. Available: https://comparch-conf.gatech.edu/dpc2/resource/dpc2_shevgoor.pdf
[42] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2015, pp. 141–152.
[43] J. E. Smith, "Decoupled access/execute computer architectures," *SIGARCH Comput. Archit. News*, vol. 10, no. 3, p. 112–119, Apr. 1982. [Online]. Available: https://doi.org/10.1145/1067649.801719
[44] A. Sodani and G. S. Sohi, "Dynamic instruction reuse," *SIGARCH Comput. Archit. News*, vol. 25, no. 2, p. 194–205, May 1997. [Online]. Available: https://doi.org/10.1145/384286.264200
[45] V. Srinivasan, R. B. R. Chowdhury, and E. Rotenberg, "Slipstream processors revisited: Exploiting branch sets," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 105–117.
[46] 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*, November 2000, pp. 257–268.
[47] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K.-M. Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen, "Helper threads via virtual multithreading on an experimental itanium® 2 processor-based platform," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XI. New York, NY, USA: Association for Computing Machinery, 2004, p. 144–155. [Online]. Available: https://doi.org/10.1145/1024393.1024411
[48] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ser. ISCA '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 2–13. [Online]. Available: https://doi.org/10.1145/379240.379246
[49] C. B. Zilles and G. S. Sohi, "Understanding the backward slices of performance degrading instructions," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, p. 172–181, May 2000. [Online]. Available: https://doi.org/10.1145/342001.339676

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on June 02,2025 at 23:32:06 UTC from IEEE Xplore. Restrictions apply.