Multi-Stream Squash Reuse for Control-Independent Processors

Qingxuan Kang
National University of Singapore
Singapore, Singapore
StarFive Semiconductor
Shanghai, China
k.qingxuan@u.nus.edu

Trevor E. Carlson
National University of Singapore (NUS)
Singapore, Singapore
tcarlson@comp.nus.edu.sg

Abstract

Single-core performance remains crucial for mitigating the serial bottleneck in applications, according to Amdahl's Law. However, hard-to-predict branches pose significant challenges to achieve high Instruction-Level Parallelism (ILP) due to frequent pipeline flushes. In typical processors, when a branch is mispredicted, subsequent instructions are indiscriminately flushed, including potentially useful instructions that will be invariably executed in the future, known as Control-Independent (CI) instructions. While existing CI proposals are effective at leveraging idiomatic control flow structures, these techniques overlook more general reconvergence scenarios. Our analysis reveals that due to the program's dynamic execution behavior, the redirected instruction stream can reconverge with not only the last squashed stream, but also several preceding streams. On average, 10% (and up to 31%) of reconvergence opportunities are overlooked if we only consider the interactions between the last squashed instruction stream and the current one.

In this paper, we introduce *Multi-Stream Squash Reuse*, which discovers execution reuse opportunities from several previously squashed streams. We use *tagged* rename mapping to enable pairwise comparisons between any two program execution states to identify data reuse. Our proposal achieves average improvements in IPC of 2.2% (SPECint2006), 0.8% (SPECint2017) and 2.4% (GAP) and maximum gains of 8.9% (*astar*), 6.1% (*bc*) and 4.0% (*cc*) from SPECint2006 and GAP benchmark suites.

CCS Concepts

• **Computer systems organization → Superscalar architectures**.

Keywords

microarchitecture, control independence, control flow reconvergence, squash reuse

ACM Reference Format:
Qingxuan Kang and Trevor E. Carlson. 2025. Multi-Stream Squash Reuse for Control-Independent Processors. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3762248

1 Introduction

Instruction-Level Parallelism (ILP) measures a processor's ability to execute multiple instructions simultaneously and make forward progress quickly. To ensure the processor backend is fully utilized, the Instruction Fetch Unit (IFU) must accurately predict the dynamic instruction stream well ahead of branch resolution, which typically occurs tens of cycles later. Therefore, tens to hundreds of instructions may enter the backend before branch resolution determines a misprediction. Branch prediction quality is crucial to ensure high pipeline utilization. However, branches whose outcomes are not dependent on program control history but are determined by data in memory are known to be hard-to-predict (H2P) by modern branch predictors [13, 16]. This can be challenging for branches whose outcomes are dynamically generated by the program, as they can be inherently random and difficult to predict with high accuracy. As the instruction windows of modern out-of-order processors grow deeper, H2P branches pose a significant bottleneck for design scaling. The inability of the IFU to predict long and accurate instruction streams can cause a significant fraction of execution cycles to be wasted on instructions that are ultimately squashed [29].

Control-Independent (CI) processors [1, 4, 5, 7, 8, 11, 15, 17, 19, 20, 23, 24, 30] complement speculative instruction fetch by mitigating branch misprediction penalties, focusing on avoiding redundant re-execution of useful work completed along the squashed path. These solutions leverage the observation that programs often branch into divergent paths and later reconverge to a common path [22]. Instructions lying on the reconverged path are known as Control-Independent instructions, which will be invariably executed regardless of the outcome of the diverging branch and may contain useful execution results. Existing CI techniques generally adopt one of two strategies: *selective flushing*, which preserves CI instructions in the pipeline and re-dispatches or repairs incorrectly executed ones; or *squash reuse*, which reuses useful execution results from the mispredicted path to the correct path. However, these approaches tend to focus on idiomatic reconvergence like *if-else* and *if-then-else* structures and do not adequately address more general forms of reconvergence where the current fetch stream may reconverge with not only the last mispredicted stream, but also any previously mispredicted streams.

In this paper, we propose *Multi-Stream Squash Reuse*, an efficient mechanism that simultaneously tracks multiple past squashed streams and performs squash reuse—low-cost reuse of execution results from the wrong path to the correct path. To identify data dependencies, we introduce *Rename Mapping Generation*, which assigns a *version ID* to each architectural-to-physical register mapping, which we call RGIDs, to track register integrity on all execution paths. Additionally, we propose hardware extensions to the

Fetch and Rename stage to perform reconvergence detection, reuse tests and memory violation checking mechanisms. We list the main contributions of the work below.

- We introduce *Multi-Stream Squash Reuse*, a general reconvergence detection and reuse scheme to allow reconvergence between non-neighboring streams.
- We highlight the challenges in existing proposals to support multi-stream reconvergence and provide quantitative comparisons with previous approaches.
- We propose our novel *Rename Mapping Generation* mechanism to check data dependencies and perform data reuse.
- We demonstrate the design advantages of our approach by comparing it with other *squash reuse* schemes.
- We evaluate the performance of our work using the gem5 simulator and assess hardware complexities by running physical synthesis on critical logic components.

The remainder of this paper is organized as follows. Section 2 provides an overview of Control-Independent (CI) architectures, analyzes the limitations of existing CI proposals and highlights how our solution addresses these challenges. Section 3 presents our microarchitectural extensions to the Fetch and Rename stages, discusses the mechanisms required to detect both register and memory dependency violations, and provides detailed design comparisons with previous work on squash reuse [5, 24, 30], highlighting the novelties and advantages of our approach. Section 4 describes our evaluation methodology and performance results, along with area, timing and power estimations for two critical logic components. We show average improvements in IPC of 2.2%, 0.8% and 2.4% on SPECint2006, SPECint2017 and GAP benchmarks and maximum gains of 8.9%, 6.1% and 4.0% on *astar* from SPECint2006, *bc* and *cc* from GAP, respectively.

2 Motivation

In this section, we first provide an overview of Control-Independent (CI) architectures, discussing their benefits and limitations. Next, we motivate the multi-stream reconvergence using a microbenchmark and compare our solution with existing techniques.

2.1 Control-Independent Processors

In a program's control flow structure, a reconvergence point is the first instruction where all paths from a branch eventually merge, also referred to as the post-dominator with respect to that branch [9]. Instructions after the reconvergence point are executed regardless of the branch outcome, forming what is known as the *Control Independent (CI)* region. Conversely, instructions before the reconvergence point form the *Control Dependent (CD)* region. If instructions in the CI regions are also *Data Independent (DI)*—i.e., their source registers are not modified in the CD regions—they become candidates for reuse, known as CIDI instructions.

Previous work has shown that a large fraction of hard-to-predict (H2P) branches reconverge within 64 instructions [6, 20]. As a result, CI architectures [1, 4, 11, 14, 20, 31] have been proposed to capture control independence by identifying CIDI instructions and applying various optimizations, which can generally be classified into two categories: *selective flushing* and *squash reuse*. The former architecture aims to avoid unnecessary re-fetching and re-execution, by holding CI instructions in the instruction window, while the latter one flushes all instructions on the wrong path but saves execution bandwidth by reusing valid results from the wrong-path execution to the correct one during re-fetching. We pursue the latter class of optimization, *squash reuse*, for efficiency and reduced complexity.

One of the first works, Dynamic Instruction Reuse [30] proposed three schemes to identify squash reuse by operand values, architectural register names and dependence chains. Register Integration [24] identifies squash reuse by physical register names and a purposeful mapping mechanism to the last data reference, called *integration*. However, these solutions store reuse information in tables, which comes with limitations including table conflicts and serialized accesses. Dynamic Control Independence [5] employs a dual-ROB design to track squashed instructions. However, only a single squashed stream can be tracked simultaneously and incurs overhead from fully associative PC matching to identify reconvergence.

2.2 Multi-Stream Reconvergence

```
// hash: A hash function generating pseuso-random number
// calc1, calc2: Compute-intensive function calls
int arr[SIZE], t0, t1, t2;
for (i = 0 ; i < SIZE; i++) {
    int data2 = hash(i);
    int data1 = hash(data2);
Br1:
    if (data1 & 0x1) {
Br2:
        if (data2 & 0x2) {
            data2 = calc1(data2);
        }
M1:
        data1 = calc1(data1);
    }
M2:
    // Potential CIDI operations
    t0 = calc2(i);
    t1 = calc2(data1);
    t2 = calc2(data2);
    arr[i] = t0 + t1 + t2;
}
```

Listing 1: Program illustrating multiple squashed streams.

In this section, we use an example microbenchmark to highlight the benefits of tracking multiple squashed instruction streams and demonstrate the advantages of our approach over two prior works on squash reuse: Register Integration [24] (table-based reuse) and Dynamic Control Independence [5] (queue-based reuse).

2.2.1 Control-Independent Data-Independent (CIDI) Operations. We use a code example shown in Listing 1 to demonstrate CIDI operations. Code label `M2` is a reconvergence point with respect to `Br1`, as both the taken and not-taken paths of `Br1` lead to the execution of code beyond `M2`. However, not every instruction after `M2` that has been executed on the squashed path can be safely reused. For example, `data1` is modified along the taken path of `Br1`. Therefore, the computation of the temporary variable `t1`, which takes `data1` as input, becomes Data-Dependent (DD) on the branch outcome of `Br1`, and its result cannot be reused. In contrast, the result of `t0` can be readily reused, as its computation depends solely

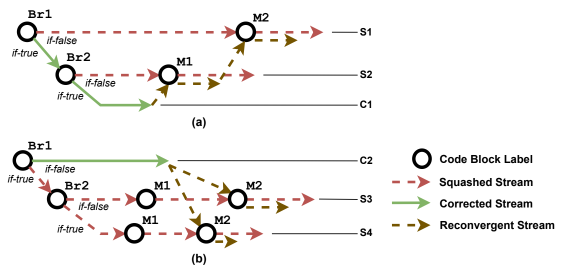

Figure 1: Illustration of Multi-stream reconvergence with respect to Listing 1. (a) software-induced multi-stream. (b) hardware-induced multi-stream.

on the loop iteration variable i, which is Data-Independent (i.e., not modified in the branch body of Br1).

In addition, t2 also presents an opportunity for reuse. When the outer branch Br1 is mispredicted but the inner branch Br2 is correctly predicted and evaluates to be not-taken, the body of Br2 does not appear on the dynamic execution stream, and data2 remains unmodified within the scope of Br1. In other words, although data2 is *statically CIDD* since it is potentially modified within the body of Br1, it can be *dynamically CIDI* under specific execution paths, hence enabling reuse. In general, the presence of CIDI instructions allows for partial reuse of computation executed on the wrong path, such as the computation for arr[i], where intermediate results t0 and t2 may be reused.

2.2.2 Multi-Stream Reconvergence Demonstration. We classify two distinct types of multi-stream reconvergence: *software-induced* and *hardware-induced*. Listing 1 illustrates a nested branch structure, consisting of an outer branch Br1 and an inner branch Br2. The inputs to Br1 and Br2 are data1 and data2, respectively—pseudo-random values generated using a hash function. As a result, both Br1 and Br2 are expected to be frequently mispredicted.

Figure 1(a) illustrates *software-induced* multi-stream reconvergence. Initially, Br1 is mispredicted, followed by the misprediction of Br2. This sequence of misprediction events results in two distinct squashed streams, S1 and S2, where S1 has fetched and executed code beyond label M2, while S2 has fetched and executed beyond label M1. Subsequently, the corrected path of Br2, denoted as C1, can reconverge onto S2 at M1, the immediate post-dominator of Br2. However, an additional opportunity for squash reuse arises from S1, the more distant misprediction. By recording only the last squashed stream, previous works will miss the second reconvergence point, resulting in suboptimal reuse. Crucially, the potential for reuse from multiple squashed streams depends on the control flow structure, which we refer to as *software-induced* multi-stream reconvergence.

Moreover, multi-stream squash reuse can also occur due to the dynamic execution behavior seen by the hardware. In modern superscalar processors, while branch prediction in the frontend is performed in order, branch resolution in the backend can occur out of order. To demonstrate out-of-order branch resolution, consider the same code example in Listing 1. We construct a scenario

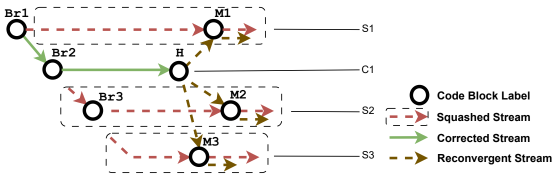

Figure 2: Overhead of maintaining poison vectors. The complete poison vector for path Br2-Br3-M3 must be partially obtained from stream S2 and S3.

where Br2 is resolved faster than Br1 in the processor backend by making data1 dependent on data2 (as shown on line 5 of Listing 1). When the result of data2 becomes available, the backend can execute Br2 and determine its direction, while data1 is still being processed, preventing Br1 from executing. Consequently, a younger, spurious branch misprediction on Br2 may occur before an older, encompassing branch misprediction on Br1.

Figure 1(b) illustrates the outcome of out-of-order branch resolution. The spurious misprediction of Br2 produces the squashed stream S3, while the overriding misprediction of Br1 generates the squashed stream S4 and the corrected stream C2. As C2 is fetched, it can reconverge with either S3 or S4, depending on which squashed stream has already executed past M2, the post-dominator of Br1. If only the most recent squashed stream, S4, is considered, reuse opportunities are lost when S3 has already reached M2 but S4 has not. Since this scenario results from the processor's dynamic execution behavior, we classify this type of multi-stream reconvergence as *hardware-induced*.

2.2.3 Comparison with Previous Work. We compare our work with two existing techniques, Register Integration (RI) [24] and Dynamic Control Independence (DCI) [5], as discussed in Section 2.1. RI utilizes a table-based approach to store squashed instructions, indexing and tagging them by instruction PC. However, a low-associativity table leads to conflicts, while high associativity quadratically increases reuse comparisons, making it timing-infeasible—a challenge identified in the original paper. The problem of table conflicts is compounded by the natural clustering of instruction footprints, as code blocks tend to occupy contiguous memory regions. When two code blocks have overlapping indexes, conflicts can extend across multiple sets, causing widespread replacements. We present the impact of table conflicts in Section 2.2.4.

On the other hand, earlier works like DCI employ a queue-based approach (dual-ROB) to store squashed instructions. However, this approach is limited to a single squashed stream, making it less effective in complex control flows that require tracking multiple squashed streams to identify reconvergence and reuse opportunities. Tracking multiple squashed streams by means of poison vectors introduces additional challenges in managing data dependencies, and solving this problem by creating additional ROB replicas is not a scalable solution. Specifically, the poison vector marks registers updated within control-dependent regions, allowing only control-independent instructions whose source registers remain unpoisoned to be reused. For example, in Figure 2, to enable the corrected stream at point H to reconverge at M1, registers along

	Nested-Mispred		*Linear-Mispred*	
	Multi-Stream Squash Reuse	Register Integration	Multi-Stream Squash Reuse	Register Integration
Single Stream / Way	2.4%	-0.1%	6.5%	1.7%
Two Streams / Ways	14.3%	1.9%	16.7%	6.2%
Four Streams / Ways	23.4%	17.9%	19.7%	16.4%

Table 1: Runtime performance improvements of two microbenchmark variations on Multi-Stream Squash Reuse and Register Integration over baseline with no squash reuse. Multi-Stream Squash Reuse enabled for 1, 2, and 4 squashed streams. Register Integration enabled for associativity of 1, 2 and 4 ways.

both the `Br1-M1` path and the `Br1-Br2-H` path must be poisoned. The challenge compounds when `H` attempts to reconverge at `M2` and `M3`, requiring additional poison vectors for paths `Br2-Br3-M2` and `Br2-Br3-M3`. Notably, for the last poison vector, register updates must be obtained partially from squashed stream `S2` and the remainder from squashed stream `S3`, necessitating inspection across multiple streams. Essentially, to ensure correct poisoning of all registers between any two diverging paths, poison vectors must be maintained not only for the diverging paths themselves, but also for the segments between neighboring branches. In the single-stream squash reuse case, there is only one branch, eliminating the need to distinguish segments within a squashed stream. In contrast, multi-stream squash reuse involves *2N* segments[1] for *N* squashed streams, significantly increasing complexity and control overhead. On the other hand, the complexity of our method remains independent of the number of squashed streams, as we will detail in Section 3.

2.2.4 *Quantitative Studies.*

To quantitatively evaluate the benefits of multi-stream squash reuse, we execute two variations of a microbenchmark derived from Listing 1. In the first variation, `Br1` depends on `data1` and `Br2` depends on `data2`. As explained in Section 2.2.2, `data1` becomes available sooner than `data2` due to data dependencies. Consequently, `Br2` triggers a branch misprediction event before `Br1`, creating a nested misprediction. We refer to this variation as *nested-mispred*. In the second variation, we swap line 8 and line 10 in Listing 1, modifying the dependencies so that `Br1` and `Br2` depend on `data2` and `data1`, respectively. In this case, branch mispredictions on `Br1` and `Br2` occur in order. We call the second variation *linear-mispred*.

To evaluate the performance benefits of tracking multiple past squashed streams, we run both microbenchmark variations on our proposed hardware solution and our implementation of Register Integration (RI). For our approach, we vary the search depth from one to four past squashed streams, each containing up to 64 instructions, to determine reconvergence and enable squash reuse. The detailed design and architecture of our proposed solution are presented in Section 3. For RI, we use a reuse table with 64 sets and vary set associativity across one-way, two-way and four-way configurations, aligning the total number of reuse entries with our work.

As shown in table 1, both microbenchmarks demonstrate performance gains beyond a single squashed stream, ranging from 2.4% to 14.3% for *nested-mispred* and 6.5% to 16.7% for *linear-mispred*. This suggests that a significant portion of squash reuse opportunities arises from non-neighboring squashed streams, highlighting an advantage over Dynamic Control Independence (DCI), which tracks only a single squashed stream. Notably, *nested-mispred* exhibits smaller gains when reusing a single squashed stream compared to *linear-mispred*, indicating that a larger fraction of reuse occurs with more distant streams. This aligns with the fact that *nested-mispred* experiences more hardware-induced out-of-order branch resolutions, whereas *linear-mispred* predominantly resolves branches in order. As a result, *nested-mispred* requires tracking more past streams to accommodate its nesting behavior, potentially across multiple loop iterations. Consequently, as the number of enabled squash streams increases from two to four, performance gains for *nested-mispred* improve further, reaching 23.4%.

Comparing our work with Register Integration (RI), we observe that RI achieves a speedup only at a higher associativity level (four way) for both microbenchmarks. Figure 3 presents the reuse table replacement frequencies, showing that low associativity results in frequent replacements, reducing opportunities for instruction reuse. At four-way associativity, fewer replacements occur, leading to speedups of 17.9% and 16.4% for the two benchmarks, respectively. However, our approach still outperforms RI, as it eliminates table conflicts and leverages reconvergence information to index into queue structures that store squashed instructions.

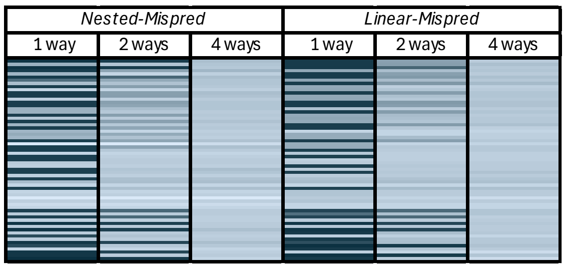

Figure 3: Replacement frequency in the RI table. Light shading indicates low replacement, dark shading indicates high replacement.

2.2.5 *Profiling Multi-Stream Reconvergence on Larger Benchmarks.*

We profile selected benchmarks from SPECint2006, 2017 and GAP to categorize different reconvergence types. We identify *software-induced* reconvergence as merging onto the squashed path of an elder branch, and *hardware-induced* reconvergence as merging onto

[1]`Br1-M1`, `Br1-Br2`, `Br2-H`, `Br2-Br3`, `Br3-M2`, `Br3–M3` in Figure 2

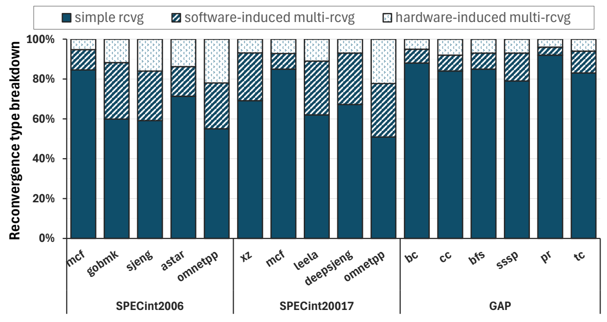

Figure 4: Breakdown of reconvergence types. A significant portion of reconvergence occurs between non-neighboring squashed streams.

the squashed path of a younger branch. When merging occurs onto the squashed path of its own diverging branch, we refer to it as *simple reconvergence*. As shown in Figure 4, most GAP benchmarks exhibit simple reconvergence. However, several SPECint2006 and SPECint2017 benchmarks with high branch misprediction rates show that a significant portion of reconverging instructions require the processor to track at least two or more squashed streams in order to perform reconvergence, as indicated by the combined fractions of software-induced and hardware-induced multi-stream reconvergence, ranging from 15% (*mcf*) to 43% (*omnetpp*).

3 Design

In this section, we present the design of our *multi-stream squash reuse* mechanism. We first explain *Rename Mapping Generation ID (RGID)*, the central component that enables multi-stream squash reuse. Next, we use a walkthrough example to illustrate the process of instruction reuse with RGIDs. Then we introduce the core architecture and the hardware extensions implemented at the Fetch and Rename stages to enable reconvergence detection and squash reuse. After that, we detail key logic components responsible for reconvergence detection and squash reuse. Finally, we address the limitations of previous work, highlighting the advantages of our approach.

3.1 Rename Mapping Generation ID

In conventional out-of-order processors, the register renaming algorithm assumes an in-order instruction stream arrival. Therefore, true dependencies can be resolved by examining if a younger instruction's source registers are referenced by any elder instructions' destination registers. However, as we revisit squashed streams that contain instructions that are yet to be executed but have been renamed in the past, identifying and maintaining correct data dependencies is critical to ensuring correct functionality.

We address this challenge by tagging every new mapping with a unique *Rename Mapping Generating ID (RGID)*. For each architectural register, we maintain a global counter that tracks the next RGID, incrementing it whenever the register is renamed. During reconvergence, the RGIDs of all source registers in the newly fetched instruction are compared with those of its squashed counterpart in the squashed stream. If all RGIDs match, it confirms that the source register values in the current execution state are identical to those from the squashed instruction's previous execution. To ensure that the execution result stored in the destination physical register remains intact, we delay freeing physical registers for executed instructions until we are certain that the instruction will not be reused[2]. Therefore, we can directly map the new instruction's destination register to the last physical register it was mapped to and set the instruction as completed immediately in the rename stage. In this scenario, it is not required to allocate a new RGID, but the RGID of the squashed instruction is forwarded to the reusing instruction, allowing subsequent instructions to observe updated RGIDs for that architectural register and perform the reuse test correctly. We show an efficient implementation of the reuse test logic in Section 3.5.

To keep track of the latest RGIDs associated with each architectural register and to support recovery, RGIDs are updated together with the *register alias table (RAT)*—a structure that maintains the most updated architectural-to-physical register mapping. RAT checkpoints are taken at regular intervals to prepare for pipeline recovery. When a pipeline flush occurs, we use a combination of checkpoint restoration and rollbacks from squashed instructions in ROB to recover the precise register mapping and RGID states.

Note that the global RGID counters are not checkpointed nor recovered, as these counters do not represent execution states, but serve the purpose of uniquely identifying mappings, both on the correct path and the wrong path.

In addition to register dependency violations, data violations can also manifest as memory order violations. To allow the reuse of load instructions, additional memory hazard tests must be performed. The detailed hardware extensions implementing this check are discussed in Section 3.8. In addition, store instructions do not update the register file or have register consumers, but their execution is crucial for detecting read-after-write violations. Therefore, we do not attempt reuse for store instructions.

3.2 Walkthrough

In this section, we use a walkthrough example to demonstrate our solution. Refer to the code snippet on the left side of Figure 5. Assume that instruction `I1` is a hard-to-predict (H2P) branch that is taken when register `t0` equals zero. The taken path of `I1` jumps to `I5` (the *if* condition) while the not-taken path falls through to `I2` (the *else* condition) and then jumps to `I7`. Instructions `I7` to `I9` form the reconvergence path, which is a CI region with respect to the branch at `I1`.

The second (middle) and third (right) tables in Figure 5 show the dynamic events generated during the execution of this snippet and the corresponding RAT/RGID updates, respectively. The RGID is updated whenever a corresponding architectural register is renamed. However, as we will demonstrate later, the RGID update policy differs between instructions reused from past executions and those newly renamed to physical registers from the Free List.

When branch instruction `I1` is encountered by the IFU the first time, it is predicted to be taken and jumps to `I5`. When `I5` is renamed, the destination register `a2` is mapped to physical register

[2]The detailed policies for freeing physical registers are discussed in Section 3.3.

PC label	Code Snippet
`I1`	`j I5 if t0==0`
`I2`	`a2 = a2 >> 1`
`I3`	`a2 = a2 + 1`
`I4`	`j I7`
`I5`	`a2 = a2 >> 2`
`I6`	`a2 = a2 - 1`
`I7`	`a1 = a1 + 1`
`I8`	`a1 = a1 >> 1`
`I9`	`a2 = a2 >> 1`

PC label	Dynamic Insn	Event
`I1`	`j I5 if t0==0`	`I1` predict taken to `I5`
`I5`	`a2 = a2 >> 2`	rename
`I6`	`a2 = a2 - 1`	rename
`I7`	`a1 = a1 + 1`	rename
`I8`	`a1 = a1 >> 1`	rename
`I9`	`a2 = a2 >> 1`	rename
		`I1` mispredicts, redirect to `I2`
`I2`	`a2 = a2 >> 1`	rename
`I3`	`a2 = a2 + 1`	rename
`I4`	`j I7`	unconditonal jump
`I7`	`a1 = a1 + 1`	reuse dst reg
`I8`	`a1 = a1 >> 1`	reuse dst reg
`I9`	`a2 = a2 >> 1`	rename

State Label	RAT with RGID			
	a1		a2	
	pn	RGID	pn	RGID
`S1`	p11	0	p22	0
`S2`	p11	0	❶ **p22→p32**	**0→1**
`S3`	p11	0	❷ **p32→p42**	**1→2**
`S4`	❸ **p11→p21**	**0→1**	p42	2
`S5`	❹ **p21→p31**	**1→2**	p42	2
`S6`	p31	2	❺ **p42→p52**	**2→3**
`S7` (Restore `S1`)	p11	0	p22	0
`S8`	p11	0	❻ **p22→p62**	**0→4**
`S9`	p11	0	❼ **p62→p72**	**4→5**
`S10`	p11	0	p72	5
`S11`	❽ **p11→p21**	**0→1**	p72	5
`S12`	❾ **p21→p31**	**1→2**	p72	5
`S13`	p31	2	❿ **p72→p82**	**5→6**

Figure 5: If-then-else example demonstrating RGID mechanisms. Left: Code snippet of the if-then-else example. Middle: Dynamic event trace executed by the processor. Right: RAT and RGID updates for each corresponding step in the middle table.

p32 at ❶, and its RGID is updated from 0 to 1. Subsequent speculative instructions I6 to I9 are renamed at ❷ through ❺, each receiving a new physical register and a new RGID. These RGIDs are obtained from a global array of RGID registers that track the next available RGID for each architectural register. Later, when a branch misprediction on I1 is triggered after the latent register result t0 is resolved, the processor is redirected to I2. At this point, the RAT and RGIDs are restored to state S1 (labeled in the first column of the right-hand table).

As the first instruction after branch misprediction, I2, reaches the Rename stage, no reusable instruction is available since the new instruction stream has not reconverged. Therefore, a new RGID is assigned at ❻. Specifically, RGID 4 is obtained by incrementing the global RGID register which was previously at 3 for register a2 at state S6. The global RGID is *not* incremented from the current value at state S8 in the RAT, as doing so would cause aliasing with S2. ❼ performs the same RGID assignment policy as ❶ to ❻.

When instruction I7 is fetched again, a reconvergence point is detected. The Rename stage compares the instructions renamed on the squashed path (from state S4 to S6) with those on the reconverged path to identify squash reuse opportunities. At ❽, the RGIDs of source register a1 are compared between states S4 and S11. Since both values are 0, this indicates that a1 has not been modified since its last execution and its data remains valid. Therefore, its previous mapping to p21 can be reused, and the RGID is updated from 0 to 1 to match S4. No new RGID assignment is required since the mapping at S11 and S4 is identical. Similarly, at ❾, the RGID of a1 at state S5 and S12 are both 1, allowing reuse of its mapping to p31.

Finally, we illustrate a case at ❿ where a data violation is detected, requiring instruction re-execution. When comparing the RGIDs of source register a2 of instruction I9 between state S6 and S13, a mismatch is identified (2 vs 5), indicating that the value of a2 has changed. Consequently, the source operand at S13 is stale, and the instruction must be re-executed. Semantically, because instruction I9 depends on I3 which is in the Control Dependent region, I9 must be re-executed.

3.3 Overall Architecture

To implement our solution for multi-stream squash reuse in hardware, we add new microarchitectural components to the Fetch and Rename stage. We add pipelined reconvergence detection logic in the Instruction Fetch Unit (IFU) to compare the instruction streams between the currently fetching one and the one that was previously squashed. We also extend the Rename stage to test for data integrity and perform squash reuse. No other significant changes are required for the rest of the processor.

3.3.1 Extensions to the Fetch Stage. During normal pipeline operations (refer to Figure 6), the IFU predicts the instruction stream at basic block granularity, such as one described in a previous work [21]. Each prediction block fetches a continuous block of instructions defined by a *start pc* and an *end pc*, where the *end pc* is always a control flow instruction with a taken branch or the instruction that reaches the 32-byte fetch limit. We assume an IFU design with an overriding predictor scheme such as the one described in XiangShan [33]. The *Fetch Target Queue* (FTQ) is an IFU structure that stores the prediction blocks and deallocates one once all branches within the block are either retired or squashed. These retired, non-speculative branches are used to train the branch predictor. We extend the functionality of the FTQ to include two new interfaces: (1) dumping squashed prediction blocks upon branch misprediction and (2) detecting an overlap between an active block and the squashed blocks to identify a reconvergence point.

When the *Branch Resolution Unit (BRU)* detects a branch misprediction during the execution stage in the processor backend, all instructions following the mispredicted branch are immediately squashed. However, instead of invalidating these younger FTQ entries after the mispredicted branch, we move these entries into *Wrong-Path Buffers* (WPBs). This two-dimensional buffer is responsible for tracking the program’s wrong path that was previously sent to the processor backend and used to compare with the corrected stream to identify the reconvergence point. Each squashed stream occupies one WPB and a round-robin replacement policy is used to select the next WPB for writing. Further analysis on choosing the optimal buffer size is presented in Section 4.

The IFU continues to monitor the new instruction stream as long as at least one WPB contains valid entries and a reconvergence point has not yet been identified. We use *Left Aligner* and *Right Aligner* logic to detect range overlaps between the new prediction block and any one of the past prediction blocks in the WPB. Detailed explanations on the aligner logic and reconvergence point detection

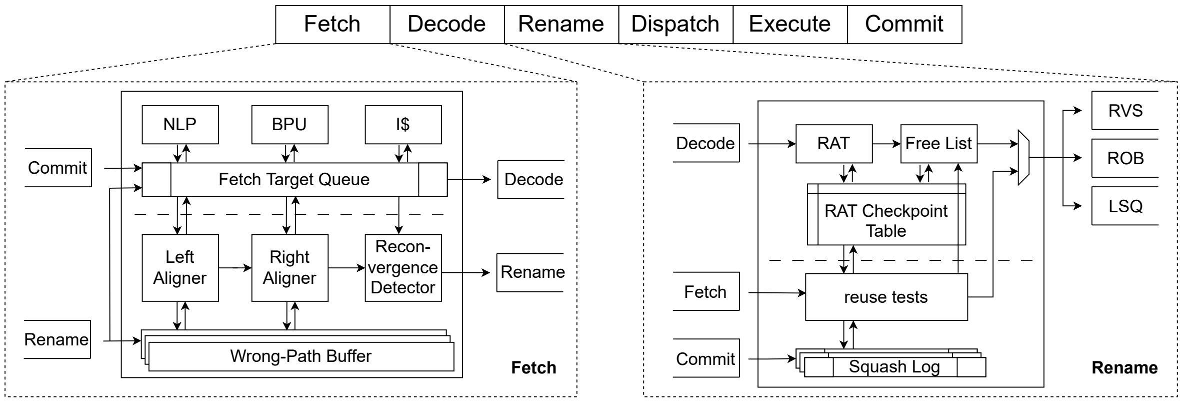

Figure 6: Overall architectural extensions (shown below the dashed lines) introduced in the Fetch and Rename stages for reconvergence detection and reuse test. NLP: Next-Line Predictor. BPU: Branch Prediction Unit. RAT: Register Alias Table.

are presented in Section 3.4. Once a reconvergence point is identified, the IFU computes the offset of the reconvergent instruction from the start of the squashed stream (one after the mispredicted branch). This offset is communicated to the Rename stage to locate the first instruction to perform the reuse test. Subsequently, the IFU monitors the fetching stream against the squashed stream to detect any divergence. When a mismatch is found, a termination signal is sent to the Rename stage to stop the reuse test. Note that the CI pipeline happens in parallel with the main fetch-decode-rename pipeline, thus incurring no extra pipeline delay.

It is possible for multiple reconvergence points to be detected within the WPBs. In such cases, we select the most recently updated WPB and the reconvergence point closest to the mispredicted branch. This strategy is chosen because the most recently squashed instruction stream is more likely to contain instructions whose source registers remain unmodified by intermediate executions.

3.3.2 Extensions to the Rename Stage. At the Rename stage, when no upcoming reconvergence is signaled by the IFU, it operates like a conventional register renaming stage. Each destination register is mapped to a new physical register that is obtained via the register Free List. Additionally, each architectural register's Rename Mapping Generation ID (RGID) is updated. The RGIDs are checkpointed together with the Register Alias Table (RAT) used for recovering from pipeline flushes. When a branch misprediction occurs, the Rename stage recovers its RAT by using a combination of checkpoint restoration and rollback [7]. Similar to the WPBs in the Fetch stage, a two-dimensional *Squash Log* is introduced in the Rename stage to facilitate squash reuse. Each stream in the Squash Log mirrors the instruction sequence of its corresponding WPB in the IFU, but operates at instruction granularity rather than fetch-block granularity. Upon a branch misprediction, the Squash Log updates a selected stream with relevant information about the squashed instruction sequence—such as execution status, source and destination RGIDs, and the destination physical register. These updates may come from both the ROB and/or the Rename stage itself, depending on the implementation of where such metadata is bookkept. Each Squash Log contains a single squashed stream and begins with the same squashed instruction as its corresponding WPB. If the length of the squashed stream is longer than the capacity of the Squash Log, younger squashed instructions are discarded. When a reconvergence point is detected by the IFU, it sends the reconvergence offset—measured from the beginning of the squashed instruction stream—to the Rename stage. This offset is used to initialize the starting position of the corresponding stream in the Squash Log for reuse tests. From this point onwards, the Squash Log operates in lockstep with the incoming instruction stream, comparing each subsequent instruction with the next entry in the Squash Log.

To prevent the physical register from being remapped to another instruction before reconvergence occurs, the physical registers occupied by the squashed (and executed) instructions are not freed immediately upon squashing but reserved unless one of the following conditions is true: (1) The squashed instruction holding on to the physical register is not yet executed by the time misprediction occurred, thus no reuse can be done. (2) No reconvergence point was detected after a fixed timeout. (3) The squashed instruction failed the reuse test and thus no reuse will occur. (4) The reconvergence stream has diverged. (5) If no free register is available in the Free List, the least recent stream's Squash Log is freed and physical registers are reclaimed. The last condition is an optimization when the processor is experiencing high register pressure, holding physical registers in the Squash Log may adversely affect performance on the correct path. We assume the physical registers can be returned to the Free List with the same bandwidth as they are allocated during renaming. Therefore, the instructing renaming process remains unaffected by any delays in register freeing due to squash reuse. Deadlock may occur if all physical registers are being held in the Squash Log, and the next incoming instruction is stalled waiting for registers to be freed. However, in practice, the Squash Log is much smaller than the number of physical registers (a total of 64 entries in our default configuration). Therefore, the system guarantees some registers will be freed and become available after the head of ROB is retired.

If no reconvergence is detected in the IFU after a fixed number of instructions (here we use 1024), the wrong-path buffer is invalidated.

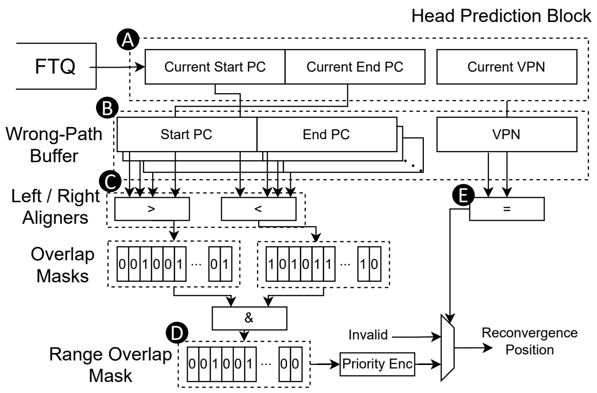

Figure 7: Architectural modifications in the Fetch stage.

The invalidation signal is also sent to the Rename stage to clear the Squash Log and free up physical registers.

A reserved *null* RGID value is used to indicate that a destination register is not reusable. This can occur when the register is non-renameable or when an RGID overflow happens. To minimize the frequency of RGID overflows and enable recovery, a global RGID reset is triggered either when all Squash Logs become unoccupied or when RGID overflow events have accumulated more than eight times during new RGID assignment. Upon a global RGID reset, the Fetch and Rename stages temporarily suspend the acceptance of new squashed streams until a fixed number of instructions—equal to the size of the ROB—have been committed. This temporary halt ensures that no squashed instruction carrying an outdated RGID (from before the reset) enters the Squash Logs and incorrectly performs a reuse test against newer RGIDs assigned after the reset.

3.4 Reconvergence Detection

In order to detect a reconvergence point in the IFU (see Figure 7), we identify the first basic block overlap between 🅐, the one currently being fetched by the IFU with 🅑, all basic blocks in the 2-dimensional Wrong-Path Buffers (WPBs) (here we show only one stream). Since each WPB entry represents a *contiguous* block of instructions—either ending with a taken branch or reaching the maximum block size of 32B—we can safely detect overlaps by comparing the *start* and *end pc* of each entry, without needing to compare individual instructions. More precisely, when the IFU produces a prediction block with a range that begins at $start\ pc_{head}$ and ends at $end\ pc_{head}$ (inclusive), we perform a fully associative check with all entries in the WPB, expressed as $start\ pc_{wpb}$ and $end\ pc_{wpb}$. As the detection logic spans multiple cycles, the critical path is not affected by this new logic. An overlap is found when the following condition is true:

$$start\ pc_{head} <= end\ pc_{wpb}\ \&\&\ end\ pc_{head} >= start\ pc_{wpb}$$

We use *left aligner* and *right aligner* modules (shown in 🅒) to evaluate the conditions above and generate two masks: one to indicate if the current *start pc* is equal to or smaller than some *end pc* in the WPB, and another to indicate if the current *end pc* is equal to or larger than some *start pc* in the WPB. These two bit-masks are bit-wise *ANDed* together to generate a final bit-mask, 🅓, indicating which entries in the WPB the current prediction block overlaps with. If multiple WPB entries overlap with the current prediction block, we take the first overlapped entry using a priority encoder. Lastly, the exact reconvergence PC is the maximum between $start\ pc_{head}$ and $start\ pc_{wpb}$ of the selected WPB entry. To further improve timing performance, we optionally restrict the WPB code span to a single physical page. The WPB records bits 12–1 of the program counter (with bit 0 always zero in the RISC-V architecture), while a single Virtual Page Number (VPN) register holds the upper 36 bits, assuming an sv48 RISC-V architecture. To detect the reconvergence point, the VPN of the current prediction block and WPB's VPN are compared in parallel with range overlap comparison to determine the final outcome, as shown in 🅔. The extra storage required for WPB in the Fetch stage is summarized in Table 2.

3.5 Reuse Test

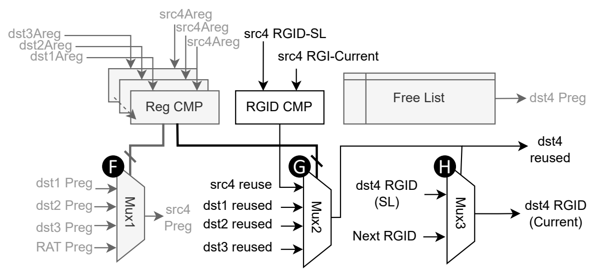

Figure 8: Reuse test for the fourth instruction in the 6-wide renaming logic. Areg: Architectural register. Preg: Physical register. SL: Squash Log.

In this section, we describe the logic that performs the reuse test at the Rename stage. To determine if an instruction can be reused, its source register RGIDs are compared pairwise with its squashed counterpart. However, as the Rename stage performs register renaming for multiple instructions per cycle, it is crucial to identify the intra-group dependencies to make sure the most up-to-date RGIDs are used for the reuse test. For example, if the second instruction depends on the first, the reuse test for the second instruction must acquire the latest RGID of the first instruction's destination register, rather than reading from the RAT. Importantly, this dependency resolution is already part of the normal register rename process. We describe how the reuse test can be performed in parallel with register renaming to avoid introducing additional critical paths. Refer to Figure 8, we show the register rename process that already existed in current processors (highlighted in grey) and our reuse test logic. For simplicity, we illustrate only the logic for processing the fourth instruction, and assume only one source register for that instruction.

At 🅕, to determine the source physical register for the fourth instruction, the Rename stage must identify the most recent physical register to which the corresponding source architectural register was renamed. This mapping may either come from one of the preceding three instructions within the same issue bundle, or from earlier cycles, with the most up-to-date mapping maintained in the Register Alias Table (RAT). To select the correct physical register among these sources, the *Reg CMP* logic compares the architectural destination registers of the preceding three instructions against the architectural source register of the current instruction to detect the most recent match. Based on this comparison, Mux1 selects the appropriate physical register for the source operand of the fourth instruction. The renaming process for other instructions is identical, except that to resolve the dependency of the n*th* instruction, we require examining the destination register for all n-1 preceding instructions, which is the critical path in the renaming logic.

In parallel, when reconvergence occurs, we perform an RGID comparison, *RGID CMP*, between the current instruction, `src4 RGID-Current`, obtained from the RAT, and its counterpart from the squash log, `src4 RGID-SL`. However, as `src4 RGID-SL` may have a more updated value from renaming previous instructions 1 to 3, we reuse the output of `Reg CMP` to select the most updated source. If the most recent instruction that writes to the same architectural register passes the reuse test, it implies a match between the RGID of the current instruction's source register and the corresponding entry in the Squash Log. This is true because the earlier instruction reuses the destination register's RGID from the Squash Log, and no subsequent instruction has modified the RGID for that architectural register. By leveraging this transitive property, the design avoids a long dependency chain—waiting for the previous instruction's RGID to become available before performing the *RGID CMP* for the current instruction. Instead, the necessary proxy information is whether previous instructions 1 to 3 have performed squash reuse successfully, as shown at Ⓖ.

Subsequently, the result of the reuse test for instruction 4 is sent as input to the rename process of the next instruction. While this introduces a dependency chain for subsequent reuse tests, its impact on timing delay will be overshadowed by the longer critical path, the *Reg CMP* in the next instructions' renaming process. This is because the next instruction must resolve an additional register dependency that spans instruction 1 through 4.

Lastly, at Ⓗ, depending on whether instruction 4 has performed squash reuse successfully, the RGID for its destination register is either reused from the Squash Log or newly allocated. RGID updates for all instructions in the issue bundle are updated to the Register Alias Table (RAT) at the end of the current cycle, along with the corresponding physical register mappings. We provide hardware synthesis results for the reuse test logic in Section 4.1.3.

3.6 Additional Storage for Architectural Extension

We summarize the additional storage requirements for the major structures in our implementation of the multi-stream squash reuse mechanism in Table 2. Both the Wrong-Path Buffer (WPB) and the Squash Log require three pointers for their operation: a *Stream Read Pointer* to read the currently active stream once reconvergence is located, an *Entry Read Pointer* to access the next entry within that stream, and a *Stream Write Pointer* to allocate the new stream. Each WPB entry stores the instruction stream's PCs at block granularity, using *Start PC* and *End PC* to define the range. Each Squash Log entry stores the RGIDs of the source and destination registers, as well as the destination physical register for each squashed instruction. The Squash Log does not store PCs, as the selected stream in both the WPB and the Squash Log corresponds to the same instruction sequence, and the IFU sends control signals to the Rename stage to indicate the beginning and end of reuse tests. The Re-Order Buffer (ROB) stores the RGIDs of all source and destination registers to support the population of the Squash Log during branch misprediction events. In our calculation, we set the ROB size to be 256 entries, matching the size of the physical register file. Additionally, to maintain RGID updates and rollback, the RAT and the RAT checkpoints are extended to include RGID states for each architectural register. In a typical configuration of 4 squashed streams (N=4), 16 WPB

Table 2: Additional storage required for squash reuse scheme. N: Number of streams; M: Number of WPB entries per stream; P: Number of Squash Log entries per stream.

Structures	Fields	Bits
Wrong-Path Buffer	Stream Read Pointer	log2(N) bits
	Stream Write Pointer	log2(N) bits
	Entry Read Pointer	log2(M) bits
	Virtual Page Number	N streams × 36 bits (PC[47:12])
	Entries	N streams × M fetch blocks
Wrong-Path Buffer Entry	Valid	1 bit
	Start PC	11 bits (PC[11:1])
	End PC	11 bits (PC[11:1])
Squash Log	Stream Read Pointer	log2(N) bits
	Stream Write Pointer	log2(N) bits
	Entry Read Pointer	log2(P) bits
	Entries	N streams × P instructions
Squash Log Entry	Valid	1 bit
	Source Register RGIDs	3 registers ×6 bits
	Destination Register RGID	1 register ×6 bits
	Destination Physical Register	1 register ×8 bits
Re-Order Buffer (ROB)	Register RGIDs	(3 source registers + 1 destination register) × 6 bits ×256 entries
Register Alias Table (RAT)	RGID per Arch-Register Mapping	64 arch-registers ×6 bits
	RAT checkpoints	64 arch-registers ×6 bits ×32 checkpoints
Additional Storage (Constant)	(4×6×256+64×6+64×6×32) = 18,816 bits = 2.30KB	
Additional Storage (Variable)	2×log2(N) + log2(M) + (1+11+11)×N×M + 36×N 2×log2(N) + log2(P) + (1+3×6+1×6 + 8)×N×P = (23×M + 33×P + 36)×N + log2(M×P×N^4) bits	

entries (M=16) and 64 squash log entries (P=64), the aggregated storage required is 3.53KB (constant: 2.30KB, variable: 1.23KB).

Note that the calculations presented in Table 2 account only for register bit storage and do not include the resources required for combinational logic. A more detailed evaluation of combinational delays, power consumption, and overall area is provided in Section 4.1.3, which includes synthesis results for the two critical logic components discussed in the previous sections: reconvergence detection in the IFU and the reuse test logic in the Rename stage.

3.7 Improvements Over Previous Work

In this section, we describe the major improvements over previous work on *squash reuse*. Our work shares the same spirit with *Dynamic Instruction Reuse (DIR)* [30], *Register Integration (RI)* [24] and *Dynamic Control Independence (DCI)* [5], as we all opportunistically reuse execution results from the squashed stream as the corrected instruction stream arrives at the Rename stage. DIR and RI stores squashed instructions in table structures but differ in their table access mechanisms, reuse metadata and maintenance policies, while both DCI and our work preserve the instruction order of the squashed stream in queue structures. We first detail three key differences of our work with DIR and RI and illustrate the limitations of storing reuse information in table formats. Next, we explain the similarities and improvements over DCI.

3.7.1 Tracking Temporal Reference. In DIR, three variations of squash reuse are proposed. Squashed results are saved in the *Reuse Buffer* and the reuse is tested using one of the three schemes, by matching every table entry with the PC and: a) input values to each source register b) source registers' architectural names c) the dynamic dependency chain. For schemes b and c, since the squashed instruction stream can have multiple dynamic instructions matching the same entry in the Reuse Buffer, only one execution result can be saved when using this previous work. Refer to the left figure in Figure 9, we say DIR cannot distinguish *temporal references* of the same architectural register at different execution contexts due to table conflicts. In the RGID method, however, temporal references are resolved by the unique Generation ID, allowing multiple execution outcomes to be tracked. Moreover, in DIR's method c, each architectural register's dependency chain is recorded using an auxiliary *Register Source Table (RST)*. However, as the RST is indexed by the architectural register, only one dependency can be tracked and there cannot be two destination registers of the same architectural name, i.e. write-after-write false dependencies cannot be resolved, limiting the opportunities for reuse. RGIDs, which form the basis of our multi-stream solution, have no such restrictions.

3.7.2 Transitive Invalidation Overhead. To ensure data integrity, both previous works, DIR and RI, require careful table maintenance. Whenever a physical destination register is remapped or a Reuse Buffer (RB) entry is evicted without being reused, any other RB entries that reference the destination register of this entry will also need to be evicted. However, the eviction of these RB entries can induce further evictions, a process we call *transitive invalidation*. Refer to the middle figure in Figure 9, in RI, when an entry is invalidated, a chain of dependent instructions will also need to be invalidated, which is an expensive operation. In our work, the invalidation is done implicitly and lazily. When an architectural register is remapped, its RGID is updated. This implicitly broadcasts the invalidation event to all entries in the Squash Log. As entries in the Squash Log undergo reuse tests, they detect their integrity and are discarded.

3.7.3 Serialized Table Access. Register Integration (RI) attempts to solve the inability to resolve temporal reference in DIR by matching the physical register names to distinguish different execution contexts. However, since physical registers are used as both values and keys for the Reuse Buffer, this results in N serialized accesses to the Reuse Buffers in highly associative implementations, a problem that was identified in the original paper. Refer to the right figure in Figure 9. In a given cycle, all N instructions may depend on their preceding instructions, and the reuse test for the current instruction can only be performed when the previous instruction has already accessed the table and acquired the physical register. RI tackles the long logical dependence by using a set-associative cache and accessing W possible reusable entries for every instruction in parallel, where W is the associativity. However, such a method is not scalable in terms of both pipeline width and Reuse Buffer (RB) associativity. Notably, RI acknowledged that the intra-dependency resolution could alternatively be handled by storing dependency chains in traces. However, concrete solutions to this challenge were deemed out of scope for their work.

3.7.4 Relationship with DCI. DCI is the first queue-based squash reuse proposal which uses a dual-ROB approach to retain squashed instruction streams. Our Squash Log in the Rename stage is similar to DCI. A key improvement over DCI is the observation that limiting reuse to only the immediate past stream can result in missed opportunities for detecting reconvergence and enabling squash reuse, when such opportunities require inspecting multiple past squashed streams. In this sense, DCI can be viewed as a special case of our approach, when the number of squashed streams to consider is one. Nonetheless, as discussed in detail in Section 2.2.3, naively extending DCI to support multiple streams introduces challenges due to the increased complexity of managing and reconstructing multiple segments of poison vectors. In contrast, our RGID scheme enables comparison between any two execution contexts without requiring knowledge of the exact execution path that led to those points. Moreover, we enhance the reconvergence detection mechanism from DCI's PC-based associative search, to a more scalable block-based range search, utilizing existing structures in the Fetch-Target-Queue (FTQ).

3.8 Handling Memory Order Violations

Besides register dependencies, memory dependencies, which are identified by memory addresses rather than register names, require dedicated hazard checking logic in addition to the use of RGIDs. We are interested in two kinds of memory order violations: (1) store-to-load hazard where a younger load obtains older data from memory before the logically preceding store performs its write to the same memory address, (2) load-to-load hazard which happens in multi-core systems, where a logical snoop request is sent in between two loads, but the younger load has already obtained older data and is not re-executed. We provide an analysis of these two cases

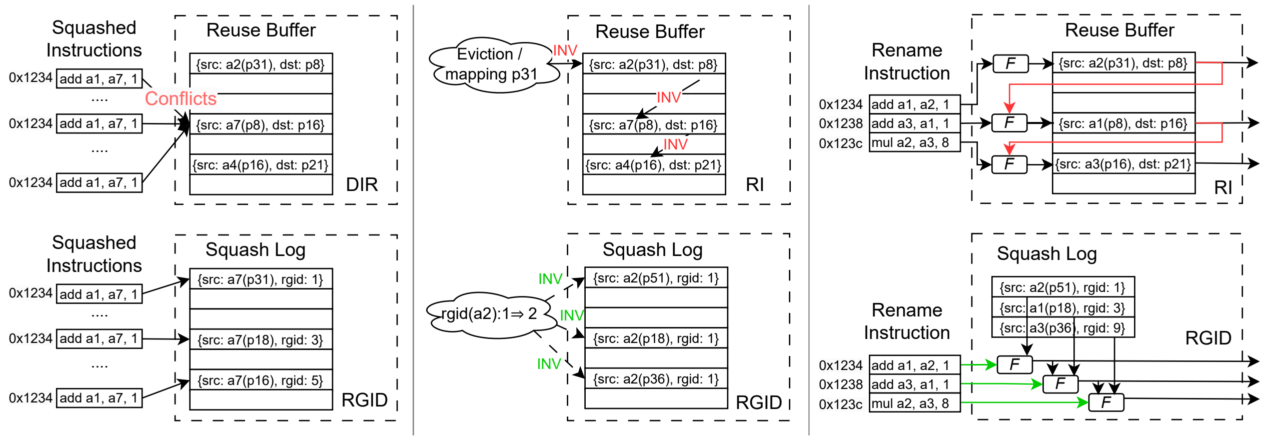

Figure 9: Comparison of DIR / RI vs RGID mechanisms. Left: Illustration of temporal reference. Mid: Illustration of transitive invalidation. Right: Illustration of serialized table access.

separately, summarize the requirements to identify these violations and propose mechanisms to fix them.

3.8.1 Store-to-Load Hazard. We assume a store-to-load hazard checking mechanism similar to XiangShan [33], where every issuing store instruction checks the Load Queue (LDQ) for any younger executed load. If a younger load is executed before an older store with an overlapping address, a store-to-load violation is detected. As a result, the load and all dependent instructions are re-executed. However, in our design, when a store is executed, the violating load might have already been squashed and no longer exists in the LDQ, thus evading the search by the store. Later, when the squashed load is bypassed, the old data is reused, causing memory violation. Therefore, we need to monitor the store addresses during the period between when loads are squashed and their reuse.

3.8.2 Load-to-Load Hazard. A load-to-load hazard occurs when a younger load is speculatively executed, and a snoop request arrives before an older load is executed, all acting on the same address. If no load-to-load checking is done, the older load would obtain more updated data than the younger load, leading to a program inconsistency. Load-to-load hazards are checked by an older load to determine if any younger executed loads have read the same address and if any targeted snoop requests have occurred since those younger loads were executed. However, in our design, since a younger completed load might have already moved to the Squash Log, we cannot track such snoop requests. Therefore, similar to store-to-load hazards, we need to additionally track the incoming snoop addresses that hit those loads in the Squash Log.

3.8.3 Additional Hazard Checking Mechanism. As described in the previous section, two additional sources need to be monitored in order to perform memory hazard checking for executed load instructions in the Squash Log: the executed store addresses and the snooped addresses. If any squashed load matches either of these addresses, it must be invalidated. Since eager invalidation is expensive, we instead propose a Bloom filter approach [25, 26]. Every interested memory address is used to update the Bloom filter that is logically maintained in the Rename stage. Subsequently, when squashed load instructions are tested for reuse in the Rename stage, their addresses (recorded in the Squash Log) are also checked in parallel against the Bloom filter. Once hit in the Bloom filter, a potential memory hazard is identified and the load must be re-executed. The Bloom filter is reset together with squash log invalidation.

An alternative solution is to re-execute all load instructions and compare their latest data with the value being reused, similar to the load verification mechanism in *NoSQ* [28]. If the values match, the load and all reused dependent instructions are considered successfully executed. Otherwise, the load has forwarded incorrect data to its dependents and a pipeline flush must be performed, together with Squash Log invalidation. In our evaluation, we choose to implement the latter mechanism for simplicity.

3.9 Compatibility with Other Extensions and Architectures

We discuss the compatibility of our design with two architectural extensions: multiple-block fetching [27] and micro-op cache [32].

3.9.1 Multiple-Block Fetching. In a multiple-block fetching scheme, the IFU predicts and fetches two instruction blocks per cycle instead of one. As a result, up to two discontiguous fetch blocks are available for processing every cycle. Since reconvergence may occur in either block, reconvergence detection must be performed for both blocks when they are fetched. To support this, our reconvergence detection logic must be duplicated, enabling both blocks to be compared with all the Wrong-Path Buffers (WPBs) in parallel, where each block is a contiguous sequence of instructions.

3.9.2 Micro-op Cache. The micro-op cache [32] is an essential component in the CISC architecture, designed to improve instruction decode throughput and reduce the power overhead of decoding complex, variable-length instructions. As modern processors adopt decoupled branch prediction and instruction fetching pipelines [18, 21], the speculative instruction stream is generated in the branch prediction pipeline—prior to the instruction fetch stage

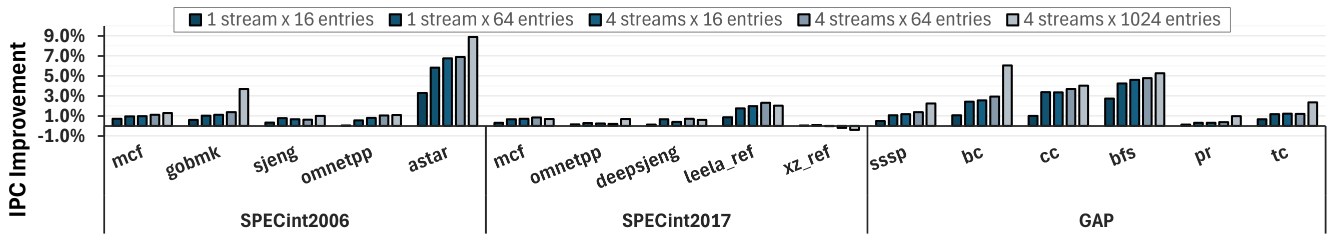

Figure 10: IPC improvements for different multi-stream configurations on SPECint2006, SPECint2017 and GAP benchmarks.

where the micro-op cache is accessed. Therefore, the reconvergence detection mechanism, which operates within the branch prediction pipeline, functions independently of the instruction fetch and decode mechanisms in the instruction fetch pipeline.

4 Evaluation

We implement our architecture in the gem5 [3] microarchitecture simulator. We use gem5's O3CPU, an out-of-order CPU model with detailed modeling of instruction Fetch, Decode, Rename, Execution, and Commit stages. We leverage gem5's execution-driven model to simulate speculative paths and model WPBs, RGIDs, and execution reuse mechanisms. The detailed CPU configuration is listed in Table 3. We select benchmarks from the SPECint2006, 2017 suites that have a branch misprediction rate of more than 3% and collect program hotspots using SimPoints [10]. The results are computed by taking the weighted sum of the cluster weights and the CPIs of each checkpoint. We also run benchmarks from the GAP [2] suite with *-g 12 -n 128*.

Table 3: gem5 Baseline Configuration.

Frontend	Fetch block size	32B
	Nextline Predictor	Bimodal
	Main Branch Predictor	TAGE-SC-L 64K
	Pipeline stages	5 stages
Backend	Decode/Rename Width	8
	Reorder buffer (ROB)	256 entries
	Reservation stations (RVS)	64-entry 4×ALU + 2×BRU + 64-entry 2×LSU
	Load-Store Queue (LSQ)	96-entry Load queue 96-entry Store queue
	Physical Register File	256 physical registers
Memory	DCache	64KB, 4-way associative, 3-cycle data latency
	L2	2MB, 8-way associative, 12-cycle data latency
	DRAM	16GBPS, 120-cycle data latency

4.1 Evaluation Results

4.1.1 Overall Performance Trend. Figure 10 shows the IPC improvements for different squash stream configurations. Major improvements are observed for *gobmk* and *astar* from SPECint2006, *leela* from SPECint2017 and most GAP benchmarks except *pr*. With a minimal configuration of 1 stream and 16 WPB entries, we observed an average speed-up of 1.0%, 0.3% and 1.0% for SPECint 2006, 2017 and GAPs, with maximum speed-ups seen on *astar* (3.3%) and *bfs* (2.7%). The relatively low performance improvement is due to WPB's inability to contain the entire squash stream at 16-entry capacity. A WPB with 64 entries strikes a good balance between performance and hardware overhead. At the 1 stream ×64 entry configuration, we see *astar* and most GAP benchmarks have doubled the IPC improvements.

To determine the appropriate number of streams to track, we analyze the *stream distance*, defined as the number of intermediate squash events between a squashed stream and the currently fetched stream at the point of reconvergence. As shown in Figure 11, over 50% of reconvergences occur between neighboring streams, and 90% to 95% occur within a distance of three streams. Based on this analysis, we configure the processor to track up to 4 streams.

We perform an upper-bound study to show the maximal gain for a large WPB configuration. In the 4 streams ×1024 entries configuration, *gobmk*, *astar* and *bc* show additional gains of 2.3%, 2.0% and 3.2% IPC improvement.

The limited performance impact observed in *mcf*, *omnetpp* and the slight negative impact in *xz* can be attributed to two factors: (1) *mcf* and *omnetpp* are highly memory bottlenecked benchmarks, hence reusing execution results provides limited benefits, as the latency reduction is overshadowed by long-latency cache miss events; (2) frequent memory order violations are observed in *xz* from SPECint2017 suite. Since our technique enables the reuse of load instructions executed far in the past, a store to the same address

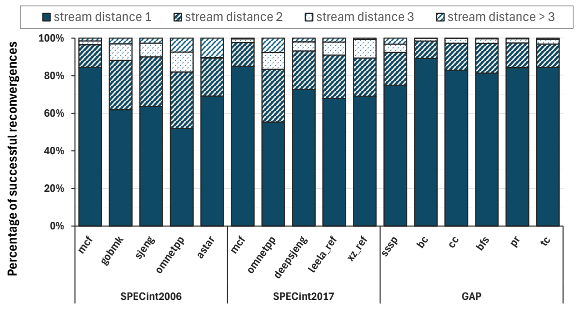

Figure 11: Reconvergence stream distance breakdown. Most reconvergence occurs within four squashed streams.

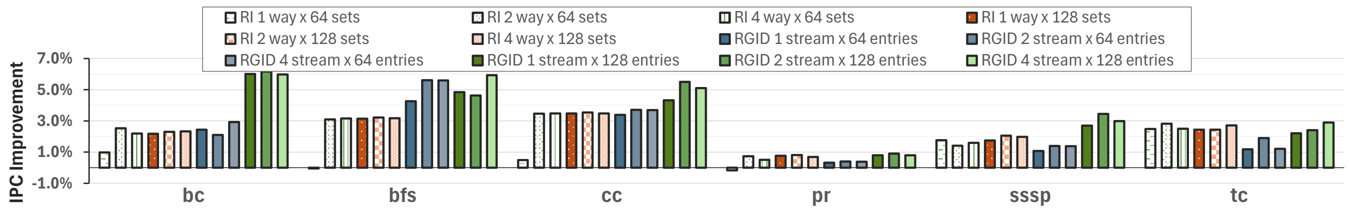

Figure 12: IPC improvements comparing Register Integration (RI) with Multi-Stream Squash Reuse solution (RGID) across different configurations on GAP benchmarks.

Table 4: Post-synthesis complexity reports.

Reconvergence Detection			
WPB Size	Logic Levels	Area / μm^2	Power / mW @ 0.7V
4×16	13	2682	1.508
4×32	19	5283	2.984
4×64	20	10369	5.909
Reuse Test (64-entry Squash Log)			
Pipeline Width	Logic Levels	Area / μm^2	Power / mW @ 0.7V
4	28	3201	3.039
6	32	4803	4.333
8	41	6256	5.509

may happen, causing the reused loads to violate memory order dependencies. Such memory violation events can incur additional pipeline flushes and degrades performance.

4.1.2 *Quantitative Comparison With Previous Work.*

We evaluate two prior *squash reuse* designs, Register Integration (RI) [24] (table-based reuse) and Dynamic Control Independence (DCI) [5] (queue-based reuse) on the GAP benchmark suite. For RI, we implement the table-based physical register indexing scheme and study various configurations of the set-associative reuse table, varying the number of ways (1, 2, and 4) and the number of sets (64 and 128). For DCI, we measure its effectiveness by configuring our multi-stream squash reuse implementation to track only a single squashed stream. To enable a fair comparison, we match hardware resources with RI in terms of the total number of squashed entries. Specifically, we vary the number of squashed streams (1, 2, and 4) and the number of entries per stream of the Squash Log (64 and 128). The size of the WPB in the IFU is set to one-fourth of the Squash Log stream size, assuming that each basic block contains four instructions on average. We call our work RGID in Figure 12 to distinguish it from RI.

As shown in Figure 12, we observe that our approach outperforms RI on *bc*, *bfs* and *cc*, while achieving comparable performance on *pr*, *sssp*, and *tc*. Moreover, our approach shows continuous gains as the stream size increases, with the most prominent improvement observed on *bc*. We find that tracking two squashed streams yields the best overall results. As deeper squashed streams are enabled for reused, the increased memory order violation events become a significant bottleneck and limit further gains. This phenomenon is consistently observed for both the RI and our RGID approach.

4.1.3 *Complexity Analysis.*

We conducted experiments on two critical logics: *reconvergence detection* and *reuse tests* using Synopsys Design Compiler and set a timing constraint at 2 GHz. The post-synthesis logic level, area and power results are summarized in Table 4. For reconvergence detection, since combinational logic is spanned across three pipeline stages, the longest logic level is only 20 even with a large WPB configuration of 4 streams and 64 entries each. The area and power scaling are mostly linear with WPB size. For the reuse test circuit, the logic level is relatively high, since resolving intra-group dependencies is an inherent critical path in superscalar processors. We identify that the critical path under the highest pressure is at RGID increments, since in the worst case, the RGID associated with an architectural register needs to be updated N times, where N is the pipeline width. We note that optimization techniques such as pre-calculation and shortening bit-width can further alleviate the pressure. Note that the ROB size has a relatively minor impact on the timing-critical path of our design. This is because Rename's Squash Log population operates in parallel with the standard Register Aliasing Table (RAT) recovery process, introducing no additional critical path.

As a reference for the total core area and power at a similar technology node used in our evaluation, *XiangShan's YXH core* [33] reported an estimated area of 3.4mm^2 with an overall power consumption of 5W[3] .

5 Conclusion

We presented a solution to *multi-stream squash reuse* by tracking multiple past squashed streams and detecting reconvergence opportunities. We analyzed a few inefficiencies with prior *squash reuse* works and proposed an efficient mechanism to reuse results executed on the wrong path in a streamlined fashion, saving execution bandwidth and shortening dependency chains. We achieved average improvements in IPC of 2.2%, 0.8% and 2.4% on SPECint2006, SPECint2017 and GAP benchmarks and maximum gains of 8.9%, 6.1% and 4.0% on *astar* from SPECint2006, *bc* and *cc* from GAP.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We also gratefully acknowledge our shepherd for the feedback and help in improving this paper.

[3]We estimated XiangShan's core area by scaling down its 8.6mm^2 die size, considering a 66% utilization rate and accounting for 60% effective core area after excluding the L2 cache footprint.

References

[1] Ahmed S. Al-Zawawi, Vimal K. Reddy, Eric Rotenberg, and Haitham Akkary. 2007. Transparent Control Independence (TCI). In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA).* 448–459.

[2] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. arXiv:1508.03619 [cs.DC]

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 Simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[4] Chen-Yong Cher and T. N. Vijaykumar. 2001. Skipper: A Microarchitecture for Exploiting Control-Flow Independence. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO).* 4–15.

[5] Yuan C. Chou, Jason Fung, and John Paul Shen. 1999. Reducing Branch Misprediction Penalties via Dynamic Control Independence Detection. In *Proceedings of the 13th Annual International Conference on Supercomputing (ICS).* 109–118.

[6] Jamison D. Collins, Dean M. Tullsen, and Hong Wang. 2004. Control Flow Optimization Via Dynamic Reconvergence Prediction. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO).* 129–140.

[7] Stijn Eyerman, Wim Heirman, Sam Van den Steen, and Ibrahim Hur. 2021. Enabling Branch-Mispredict Level Parallelism by Selectively Flushing Instructions. In *Proceedings of the 54th International Symposium on Microarchitecture (MICRO).* 767–778.

[8] Amit Gandhi, Haitham Akkary, and Srikanth T. Srinivasan. 2004. Reducing Branch Misprediction Penalty via Selective Branch Recovery. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA).* 254–264.

[9] Rajiv Gupta. 1992. Generalized Dominators and Post-Dominators. In *Proceedings of the 19th Symposium on Principles of programming languages (POPL).* 246–257.

[10] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and More Flexible Program Phase Analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[11] Andrew D. Hilton and Amir Roth. 2007. Ginger: Control Independence Using Tag Rewriting. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA).* 436–447.

[12] Qingxuan Kang. 2025. *Artifacts for "Multi-Stream Squash Reuse for Control-Independent Processors".* https://doi.org/10.5281/zenodo.17021834

[13] Chit-Kwan Lin and Stephen J. Tarsa. 2019. Branch Prediction Is Not A Solved Problem: Measurements, Opportunities, and Future Directions. In *Proceedings of the 20th International Symposium on Workload Characterization (IISWC).* 228–238.

[14] Kshitiz Malik, Mayank Agarwal, Sam S. Stone, Kevin M. Woley, and Matthew I. Frank. 2008. Branch-Mispredict Level Parallelism (BLP) for Control Independence. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture (HPCA).* 62–73.

[15] Vignyan Reddy Kothinti Naresh, Rami Sheikh, Arthur Perais, and Harold W. Cain. 2018. SPF: Selective Pipeline Flush. In *Proceedings of the 36th International Conference on Computer Design (ICCD).* 152–155.

[16] Celal Öztürk and Resit Sendag. 2010. An Analysis of Hard To Predict Branches. In *Proceedings of the 10th International Symposium on Performance Analysis of Systems and Software (ISPASS).* 213–222.

[17] Alex Pajuelo, Antonio González, and Mateo Valero. 2005. Control-Flow Independence Reuse via Dynamic Vectorization. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS).* 10pp.

[18] Arthur Perais, Rami Sheikh, Luke Yen, Michael McIlvaine, and Robert D. Clancy. 2019. Elastic Instruction Fetching. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA).* 478–490.

[19] Vlad Petric, Anne Bracy, and Amir Roth. 2002. Three Extensions to Register Integration. In *Proceedings of the 35th International Symposium on Microarchitecture (MICRO).* 37–47.

[20] Nathanaël Prémillieu and André Seznec. 2012. SYRANT: SYmmetric Resource Allocation on Not-Taken and Taken Paths. *ACM Transactions on Architecture and Code Optimization (TACO)* 8, 4 (2012), 43:1–43:20.

[21] Glenn Reinman, Todd M. Austin, and Brad Calder. 1999. A Scalable Front-End Architecture for Fast Instruction Delivery. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA).* 234–245.

[22] Eric Rotenberg, Quinn Jacobson, and James E. Smith. 1999. A Study of Control Independence in Superscalar Processors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA).* 115–124.

[23] Eric Rotenberg and James E. Smith. 1999. Control Independence in Trace Processors. In *Proceedings of the 32th International Symposium on Microarchitecture (MICRO).* 4–15.

[24] Amir Roth and Gurindar S. Sohi. 2000. Register Integration: A Simple and Efficient Implementation of Squash Reuse. In *Proceedings of the 33rd International Symposium on Microarchitecture (MICRO).* 223–234.

[25] Elham Safi, Andreas Moshovos, and Andreas G. Veneris. 2008. L-CBF: A Low-Power, Fast Counting Bloom Filter Architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 6 (2008), 628–638.

[26] Simha Sethumadhavan, Rajagopalan Desikan, Doug Burger, Charles R. Moore, and Stephen W. Keckler. 2003. Scalable Hardware Memory Disambiguation for High ILP Processors. In *Proceedings of the 36th International Symposium on Microarchitecture (MICRO).* 399–410.

[27] André Seznec, Stéphan Jourdan, Pascal Sainrat, and Pierre Michaud. 1996. Multiple-Block Ahead Branch Predictors. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).* 116–127.

[28] Tingting Sha, Milo M. K. Martin, and Amir Roth. 2006. NoSQ: Store-Load Communication Without A Store Queue. In *Proceedings of the 39th International Symposium on Microarchitecture (MICRO).* 285–296.

[29] Kevin Skadron, Pritpal S. Ahuja, Margaret Martonosi, and Douglas W. Clark. 1999. Branch Prediction, Instruction-Window Size, and Cache Size: Performance Trade-Offs and Simulation Techniques. *IEEE Trans. Comput.* 48, 11 (1999), 1260–1281.

[30] Avinash Sodani and Gurindar S. Sohi. 1997. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA).* 194–205.

[31] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar Processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA).* 414–425.

[32] Baruch Solomon, Avi Mendelson, Doron Orenstein, Yoav Almog, and Ronny Ronen. 2001. Micro-Operation Cache: A Power Aware Frontend For the Variable Instruction Length ISA. In *Proceedings of the 6th International Symposium on Low Power Electronics and Design (ISLPED).* 4–9.

[33] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, Zuojun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Quan, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. 2022. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *Proceedings of the 55th International Symposium on Microarchitecture (MICRO).* 1178–1199.

A Artifact Appendix

A.1 Abstract

The artifacts include a simulation model, workloads and utility scripts. The simulation model is a port of the open-source gem5 infrastructure, extended with our *Multi-Stream Squash Reuse* implementation and a previous work, *Register Integration.* For workloads, the GAP benchmark suite and a partial set of gem5-compatible SPEC2006int and SPEC2017int checkpoints (generated via the SimPoint methodology) are provided to evaluate the effectiveness of our solution. In addition, build scripts, run scripts, and post-processing utilities are provided to facilitate experiment execution and to collect and compare results across different configurations.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Cycle-accurate microarchitecture model
- **Programs:** GAP, SPECint2006, SPECint2017
- **Compilation:** gcc, g++
- **Run-time environment:** Linux (no root access required)
- **Hardware:** x86-64 / amd64 processors (64+ cores recommended)
- **Metrics:** Simulated runtime
- **Output:** CSV tables
- **Experiments:** Provided scripts
- **Disk space required (approx.):** 5 GB
- **Time to prepare workflow (approx.):** 1 hour
- **Time to complete experiments (approx.):** 6 hours
- **Publicly available?:** Yes
- **Code license (if publicly available):** MIT License

A.3 Description

A.3.1 How to access. All data required to perform the artifact evaluation can be found at Zenodo [12].

A.3.2 Hardware dependencies. An x86-64 / amd64 server, preferably with 64 or more hardware threads.

A.3.3 Software dependencies. We use `Apptainer` as the container platform to manage all system dependencies. Please ensure that `Apptainer` is installed on your system before proceeding.

A.3.4 Data sets. The workloads are included in the Artifact Evaluation tarball.

A.4 Installation

Place `ae.tgz` in the current working directory and run:

```
$ tar xvf ae.tgz
```

The decompressed archive will have the following directory structure:

```
ae/
 ├── sandbox.sif
 ├── scripts/
 │    ├── build.sh
 │    ├── clean.sh
 │    ├── env.sh
 │    ├── rollup_gapbs.py
 │    ├── rollup_spec.py
 │    ├── run_gapbs.sh
 │    └── run_spec_cpts.sh
 └── work/
      ├── gapbs/
      ├── gem5/
      ├── spec_cpts/
      └── tools/
```

A brief explanation of the file structure is as follows:

- `sandbox.sif`: the container image managed by `Apptainer`.
- `scripts`: build scripts, run scripts, and post-processing Python utilities.
- `work`: workloads, the gem5 model, and precompiled tools used for compilation.

Next, enter the container environment by:

```
$ cd ae
$ apptainer run sandbox.sif
```

You should see the following output:

```
Welcome to your Ubuntu container!
```

To build all dependencies (gem5, GAP benchmarks), run:

```
$ source scripts/env.sh
$ source scripts/build.sh
```

A.5 Experiment workflow

To run the GAP and SPEC checkpoints, run:

```
$ source scripts/run_gapbs.sh
$ source scripts/run_spec_cpts.sh
```

This should take approximately 6 to 12 hours, depending on the number of hardware threads available.

A.6 Evaluation and expected results

The raw outputs of the experiments are stored in the top-level `output/` directory. To gather the results, run:

```
$ python3 scripts/rollup_gapbs.py
$ python3 scripts/rollup_spec.py
```

An example output should look similar to the following:

```
          CFG    BM      CYCLES      diff
1    RCVG_4_64   bfs  76244487.0  0.050558
2    RCVG_1_16   bfs  79171252.0  0.011721
3    RCVG_2_64   bfs  76303937.0  0.049739
...
```

The four columns are explained as follows:

- `CFG`: Configuration of the experimental setup.
- `BM`: Benchmark being evaluated.
- `CYCLES`: Simulated runtime, measured in cycles.
- `diff`: Improvement in runtime relative to the baseline.

A.7 Experiment customization

Our `run_gapbs.sh` and `run_spec_cpts.sh` scripts allow customization of multiple parameters of the Wrong-Path Buffer (WPB) for *Multi-Stream Squash Reuse* and the Reuse Table for *Register Integration.* For further details, please refer to the arguments of the `run` function within each script.