Speculative Code Compaction: Eliminating Dead Code via Speculative Microcode Transformations

Logan Moody*, Wei Qi*, Abdolrasoul Sharifi*, Layne Berry*, Joey Rudek*, Jayesh Gaur†,
Jeff Parkhurst‡, Sreenivas Subramoney†, Kevin Skadron*, and Ashish Venkat*
*Department of Computer Science, University of Virginia
{lgm4xn, wq4sr, as3mx, vlb9ae, jer5ae, skadron, venkat}@virginia.edu
†Processor Architecture Research Lab, Intel Labs
{jayesh.gaur, sreenivas.subramoney}@intel.com
‡Intel Corporation
jeff.parkhurst@intel.com

***Abstract*—The computing landscape has been increasingly characterized by processor architectures with increasing core counts, while a majority of the software applications remain inherently sequential. Although state-of-the-art compilers feature sophisticated optimizations, a significant chunk of wasteful computation persists due to the presence of data-dependent operations and irregular control-flow patterns that are unpredictable at compile-time. This work presents *speculative code compaction* (SCC), a novel microarchitectural technique that significantly enhances the capabilities of the microcode engine to aggressively and speculatively eliminate dead code from hot code regions resident in the micro-op cache, and further generate a compact stream of micro-ops, based on dynamically predicted machine code invariants. SCC also extends existing micro-op cache designs to co-host multiple versions of unoptimized and speculatively optimized micro-op sequences, providing the fetch engine with significant flexibility to dynamically choose from and stream the appropriate set of micro-ops, as and when deemed profitable.**

SCC is a minimally-invasive technique that can be implemented at the processor front-end using a simple ALU and a register context table, and is yet able to substantially accelerate the performance of already compile-time optimized and machine-tuned code by an average of 6% (and as much as 30%), with an average of 12% (and as much as 24%) savings in energy consumption, while eliminating the need for profiling and offering increased adaptability to changing datasets and workload patterns.

***Keywords*-Microarchitecture, Speculation, Optimization**

I. Introduction

The slowdown of Moore's law has driven the microprocessor industry to a formidable inflection point that has been characterized by consistently diminishing rates of improvement in the execution efficiency of modern general-purpose processors, which continue to drive a substantial chunk of the world's computational demands. The software landscape, on the other hand, has evolved rapidly over the years, resulting in the emergence of complex workloads with vastly diverse execution characteristics, exposing the already widening semantic gap between the executable code and the execution hardware.

Modern workloads, such as graph, media, and healthcare analytics, increasingly feature data-dependent computation and control flow patterns that often vary considerably with changing datasets and configuration parameters [1], [2], [3], [4]. This has greatly reduced the effectiveness of conventional compiler optimizations that rely on hoisting program invariants identifiable at compile-time to eliminate redundant computation and further apply machine-specific optimizations to the residual code. Run-time optimizations [5], [6], [7], [8], [9], on the other hand, have largely been profile-guided and conservative in nature, significantly limiting their ability to adapt to changing execution profiles as datasets evolve, notwithstanding high profiling overheads and deployment costs. These obstacles have highlighted the pressing need for systems that can aggressively and yet seamlessly optimize machine code, adapting to the execution environment, even post-compilation and post-deployment.

The key to this research is the observation that modern datasets offer remarkable predictability, because data access and control flow patterns in many workload-dataset pairs manifest as invariants across long execution intervals, sometimes even spanning multiple program phases, unmasking a heretofore untapped pocket of opportunity for speculatively optimizing such code paths at run-time. This paper presents *speculative code compaction* (SCC), a novel scheme for speculatively optimizing machine code in execution to expose and eliminate dead code, entirely at runtime within the processor, based on dynamically predicted data and control invariants, thereby enabling the continuous optimization of inherently sequential code.

SCC is a prediction-driven microarchitectural technique that advances state-of-the-art dynamic binary optimization schemes in several important ways. First, it leverages the rich contextual knowledge available from a wide array of in-processor speculation techniques such as branch prediction, address prediction, and value prediction to track deterministic computational patterns in hot code regions and further transform an instruction sequence in execution into a compact stream of speculatively optimized instructions. Second, it opens the door to the deployment of aggressive and speculative dead-code elimination techniques that have been traditionally deemed unsafe, thanks to the processor's ability to rollback execution to a safe point in the event of a misprediction by flushing speculatively optimized instructions and redirecting execution to the corresponding stream of unoptimized instructions. Third, by exploiting predictability as the basis for optimization, it offers seamless adaptability to changing datasets and workload patterns, unlike existing schemes that either require extensive profiling or tend to make conservative assumptions when they lack adequate context about the dynamic execution behavior.

More importantly, SCC is a minimally-invasive, hardware-only strategy that neither requires the construction and analysis of control-flow graphs or dynamic execution traces, nor entails the application of multiple passes of sophisticated optimizations, and is yet able to look ahead multiple hot basic blocks (e.g., those that are resident in the micro-op cache) to eliminate dead code in one single pass. In fact, by extending the processor's microcode engine to include a simple ALU and a small architectural register file to track data invariants, SCC is able to apply a suite of standard peephole optimizations, including constant folding, constant propagation, branch folding, and dead code elimination, in a single pass, to speculatively transform a given sequence of hot micro-ops into a more compact stream.

Due to the inherently speculative nature of our technique, its profitability depends greatly upon the number and range of speculatively identified data/control invariants, the confidence in our predictions, and the overall coverage we are able to achieve. Relying on only high confidence predictions might yield fewer speculative invariants, resulting in low overall code compaction. On the other hand, liberal use of invariants predicted with low confidence might potentially allow for more aggressive code compaction, but also significantly increases the risk of squashing due to frequent mispredictions. One of the major contributions of this work is an extensive profitability analysis to identify an appropriate set of target code regions that are amenable to run-time speculative transformations, along with their associated predicted invariants that yield the maximum gains. As a by-product of this analysis, we arrive at several interesting insights that redefine conventional branch and value prediction design practices, in the renewed context of speculative code compaction.

Furthermore, to avoid the repeated cost of applying our speculative transformations on the same code region, we propose novel extensions to existing micro-op cache designs to co-host multiple (unoptimized and speculatively optimized) versions of micro-op sequences, enabling the processor front-end to dynamically choose from alternate versions based on the profitability analysis, and further stream the appropriate set of instructions out of the micro-op cache. There are substantial benefits to providing the front-end with this choice and flexibility. First, it allows us to gracefully recover from mispredictions by seamlessly switching to the appropriate unoptimized micro-op sequence in the micro-op cache, when available. Second, it allows us to cater to oscillating data and branch access patterns by appropriately chaining together multiple versions of speculatively optimized micro-op sequences. Third, it equips us with the ability to quickly react to changes in the dynamic execution behavior (e.g., reaching the end of a hot loop) where a predicted invariant may no longer continue to hold.
In summary, we make the following major contributions:

- We introduce SCC, a novel microarchitectural technique that has the ability to leverage speculatively identified program invariants to accelerate inherently sequential and statically compile-time optimized code, without the need for profiling, compiler metadata, or other source-level information.
- Our technique is minimally invasive and can be implemented completely within the processor front-end at less than 1.5% in area overhead.
- We propose novel micro-op cache extensions to co-host unoptimized and speculative optimized instruction sequences, providing the front-end with significant flexibility to choose from and stream the most profitable instruction sequence at run-time.
- We conduct extensive profitability analyses and sensitivity studies to understand the impact of speculative processor features such as branch and value predictors in the context of speculative code compaction.
- Overall, we achieve an average speedup of 6% and as much as 30%, while saving an average of 12% on average and as much as 24% in energy consumption.

II. Related Work

In this section, we discuss relevant literature that is most closely related, and further elucidate the distinguishing aspects of the proposed work.

Dynamic Binary Optimization. Multiple software-based dynamic binary instrumentation frameworks, such as PIN [10], Valgrind [11], and DynamoRio [12], have been proposed for program inspection, shepherding, security hardening, performance analysis, binary lifting, and dynamic optimizations among other applications [5], [6], [7], [8], [9], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]. These techniques rely on heuristics and profiling information to identify data and branch access patterns useful for dynamic optimizations and machine-specific tuning. While offline profiling requires multiple representative runs of the application to accurately capture stable, representative execution characteristics, online profiling may be limited to certain hot-code regions to amortize the high cost of dynamically probing performance counters and other management overhead, resulting in low coverage for many applications. These approaches also typically rely on a software-based code cache to store optimized code regions that are sufficiently hot, thereby avoiding the repeated cost of run-time optimization. However, unlike SCC, these approaches lack the flexibility of automatically switching back and forth between unoptimized and different speculatively optimized versions of the code, seamlessly adapting to changing workload patterns.

Superoptimization. Massalin [23] first introduced the idea of *superoptimization* – a technique that can further reduce a compile-time optimized machine code sequence into a functionally equivalent, but more compact and optimal instruction sequence, greatly enhancing the quality of the generated code. Since then, a number of superoptimization [24], [25], [26], [27], [28], [29], [30], [31], [32], [33] strategies have been described, with some relying on classical peephole rules (e.g., constant folding, simple if-conversion, redundant load elimination, etc.), and some others automatically generating rules for optimization via stochastic methods, implemented using standard static and dynamic binary translation techniques. While SCC shares their goal of optimizing already compile-time optimized machine code, the microcode transformations that it deploys are aggressively speculative and completely realized in hardware, and more importantly, do not require functional equivalency tests.

Trace Caches and Trace Processors. Rotenberg, et al. [34] introduced trace caches, which augment the processor's front-end with the ability to construct and cache long streams of dynamic instructions called *traces* that span multiple hot basic blocks that are not necessarily contiguous in the instruction cache, implicitly predicting or "folding" branches and thereby greatly improving the fetch throughput. They further propose trace processors [35] that allow for the parallel scheduling and execution of multiple independent traces as a unit, opening the door to a variety of ILP-enhancing optimizations [36], [37], [38], [39], [40], [41], including instruction scheduling, move elimination, constant propagation, and collapsing dependency chains.

Despite their potential benefits, trace processors tend to suffer from inefficiencies that arise due to management and storage overheads. In particular, trace-based optimizations have been shown to be profitable only when multiple independent execution traces that are sufficiently long are made available for optimization. However, capturing and caching long traces that don't overlap is known to incur non-trivial overheads in terms of latency, power, and hardware complexity [42], [43], [44], [45]. The effectiveness of trace caches and trace processors is also highly dependent on their ability to accurately predict branch directions and confidence, in order to construct stable traces [46]. Although Intel's Pentium 4 based on the Netburst microarchitecture featured trace caches, they were discontinued in subsequent generations and have since been replaced by simpler micro-op caches [47] used to store decoded micro-ops, eliminating the repeated cost of decoding complex x86 instructions into internal micro-ops when cached translations are available. Micro-op caches have been staple feature of Intel processors since the Sandy Bridge microarchitecture.

SCC differs from trace processors in several key ways. First, SCC is a lightweight, frontend-only enhancement geared toward supplying the execution engine with a compact instruction stream that is devoid of dead code, unlike trace processors, whose primary goal is to extract and exploit greater instruction-level parallelism by creating, scheduling, and executing multiple independent traces in parallel using dedicated trace processing engines. Second, unlike trace processors, SCC does not construct long execution traces, but simply generates a more compact version of decoded micro-ops that are already resident in the micro-op cache, thereby considerably alleviating concerns related to creation, storage, and management of long traces. Third, while trace processors also rely on value prediction to predict live-ins to individual traces for enabling trace-level parallelism, SCC takes a general and more aggressive approach by using high-confidence predictions for potentially every instruction during the compaction process. This allows for greater code compaction even for highly optimized and machine-tuned code, unlike trace-based optimizations whose effects tend to wane as the level of optimization increases [48]. Fourth, by allowing the fetch engine to choose from multiple versions of optimized and unoptimized micro-op streams, SCC provides significant flexibility by enabling speculative optimizations only when deemed profitable.

Decode- and Rename-Time Code Customization. Code-morphing architectures such IBM's DAISY [30], Transmeta's Crusoe [49], and Nvidia's Tegra K1 Denver [50] feature a software- or firmware-based binary translation layer to dynamically translate and optimize machine code from one instruction set architecture to another. Further, several other lightweight schemes for editing and customizing dynamic instruction streams have been proposed in the literature for a wide range of applications including silent store elimination, reference combining, bounds checking, code decompression, debug support, and on-demand de-vectorization [51], [52], [53], [54], [55], [56], [57], [58], [59], [60]. In addition, Kotra, et al. [61] demonstrate that micro-op cache is fragmented due to terminating conditions and propose a Cache Line boundary AgnoStic uoP cache design (CLASP) and compaction to address the fragmentation. While some of these schemes are deployed at the microcode engine, none of them take advantage of data/control predictability to transform decoded micro-op sequences into a more compact stream of speculatively optimized micro-ops, and most of them introduce greater hardware complexity and power/area overhead, in comparison to SCC that incurs just 1.5% in area overhead.

Furthermore, register renaming in modern out-of-order superscalar processors offers an important avenue for optimizing away redundant operations. This has been exploited in multiple prior academic works [62], [63], [64] to perform move elimination, reassociation, constant propagation, redundant-load elimination, and silent-store elimination, among other optimizations. Recent Intel microarchitectures feature several rename-time optimizations including move elimination and zero/one idiom evaluation. However, the key difference here is that SCC applies its transformations based on dynamically-predicted program invariants rather than the current register state, allowing for the exploitation of a larger window of opportunity, resulting in a substantially greater reduction in the overall dynamic instruction count. In addition, EOLE [65] proposes an early execution scheme such that instructions whose operands are immediate or predicted can be executed in place, in-order without being fed into the out-of-order engine. Finally, Perais [66] shows that rename-time move elimination and zero/one idiom evaluation may be leveraged to speculatively eliminate instructions whose operands may be predicted. While SCC also leverages value prediction to speculatively identify data invariants and eliminate instructions, it does so earlier in the pipeline by tracking hot code regions in the micro-op cache and employs a wider range of optimizations.

Event-Driven Optimization. The literature also describes several hardware-based event-driven and lead-follower approaches to dynamically optimize machine code in execution. These approaches typically leverage a spare hardware context to deploy a helper thread whose goal is to accelerate a given main thread in execution by targeting a wide range of low-level optimizations including prefetching, folding delinquent branches, eliminating redundant computation along forward load slices, and speculative parallelization of hot loops [67], [68], [69], [70], [71]. In contrast, SCC does not need a spare hardware context and is able to apply all of its speculative transformations in a single pass.

Transactional Memory. Transactional Memory (TM) [72] introduced the idea of executing code as a sequence of atomic transactions, such that each transaction is validated before committed, and if validation does not pass, the transaction is aborted with no side effects and re-executed. While Hardware Transactional Memory (HTM) entails significant additional hardware complexity, Software Transactional Memory (STM) [73] can be implemented using first class C/C++ constructs [74] to block non-transactional accesses to a memory location being accessed by an unfinished transaction. However, software-based isolation barriers might incur a high performance penalty, calling for sophisticated barrier optimization techniques [75].

TM can also be used to support Thread Level Speculation (TLS) [76], [77], [78], [79], a runtime technique that allows multiple parts of sequential code to be run in parallel by opportunistically assuming that they can be run in parallel and rolling back upon violations. Our work falls into the same general realm of dynamic optimization techniques, but TM has not been applied to instruction-level optimizations such as the ones discussed in this paper. Further, although TM semantics include the ability to roll back state due to instructions already retired from the reorder buffer, they may be too heavyweight for tasks such as code compaction. It is, however, an interesting direction for future work.

III. Architectural Overview

The key motivation behind this research is that modern workloads spend a substantial chunk of their lifetime executing a small number of hot-code regions, characterized by predictable computational patterns that tend to manifest as invariants over long execution intervals.

Modern processor front-ends and microcode engines have the ability to not only track and cache hot code regions in on-chip buffers such as the micro-op cache, but also provide valuable snapshots of predictable computational patterns through speculative features such branch and value predictors. In this section, we provide a detailed architectural overview of SCC that exploits and extends these fundamental processor features to realize substantial gains in single thread performance and efficiency.

Speculative Code Compaction. The central element of our architecture is a *speculative code compaction unit* (shown in Figure 1) that leverages these fundamental processor features to analyze and transform hot micro-op sequences in the micro-op cache, spanning as many as three micro-op ways within a set (roughly 18 fused micro-ops or a 32-byte native x86 code region), into one or more compact and speculatively optimized micro-op sequences. In particular, when a micro-op cache line reaches a preset hotness threshold, a code compaction request is initiated by the processor front-end and is queued up in a request queue that is appropriately sized based on the fetch width of the processor. Our experiments indicate that, on a processor modeled after Intel's Icelake architecture, even a request queue with as low as 6 entries is capable of identifying several hot code regions amenable for speculative code compaction.

In the next cycle, the request is dispatched to the SCC unit, if available, at which point, it speculatively transforms the micro-op sequence at the requested address, processing one instruction every cycle, based on dynamically predicted program invariants leveraging hints from in-processor features such as the branch predictor, loop stream detector, and value predictor. The SCC unit itself includes: (1) a register file to track speculatively identified live integer and condition-code registers, and (2) a simple integer ALU to evaluate and speculatively eliminate dead code. Due to the front-end placement of the integer ALU, we take a conservative latency/power-sensitive approach by restricting the range of operations it can perform to only simple integer arithmetic, logic, and shift operations; as a result, the SCC unit in our implementation forgoes optimization of floating-point arithmetic, loads and stores, and complex integer operations such as multiply and divide, but this would be an interesting area for future work. However, despite this restriction, we observe substantial reductions in the dynamic instruction

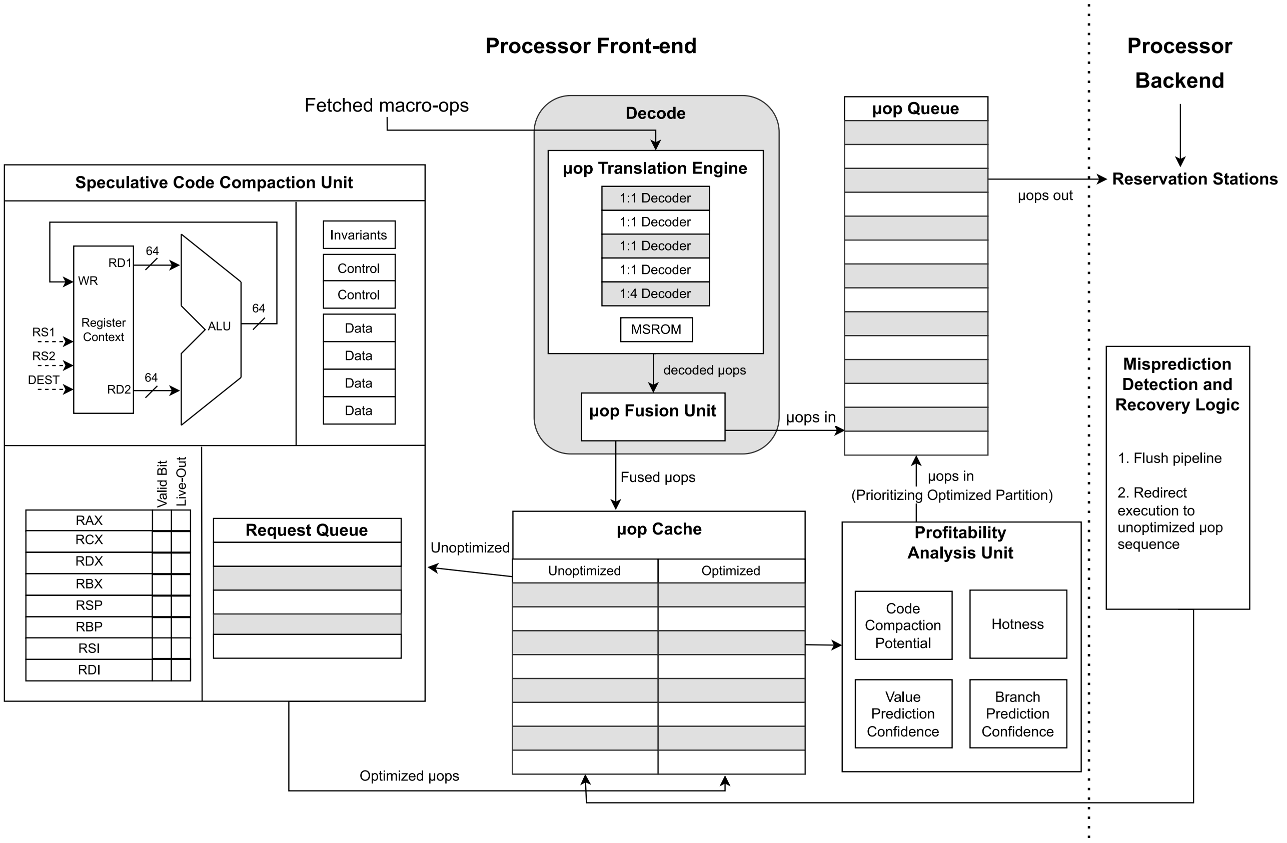


Figure 1: Architectural Overview of Speculative Code Compaction

count for most applications, including some that feature intense floating-point activity, due to the sheer ubiquity of code regions that use integer arithmetic and logic operations.

The SCC unit also interfaces with predictor units in the processor to identify potential data and control invariants that can be confidently predicted. In our experiments, we observe that most 32-byte code regions are small enough that they tend to use no more than four data invariants and two control invariants that are speculatively identified. Based on the type of the micro-op being processed and the set of data and control invariants, the SCC unit is able to either evaluate and speculatively eliminate the micro-op leveraging the ALU or speculatively perform a suitable addressing mode transformation on it, such as constant propagation by conversion from register-register to register-immediate format. The transformed micro-op is then placed in an appropriately sized write buffer (can store 18 micro-ops in our implementation modeled after Icelake), so that the SCC unit can proceed to the next micro-op in sequence in the next cycle.

Since the SCC unit operates in parallel with other existing fetch logic that probes the predictors, we double the prediction width (along with the associated logic) to allow the fetch engine to simultaneously read two predictor entries at once. We model this in CACTI [80] and report the associated area and peak power overheads in Section VII. We also note that a single copy of the prediction histories is maintained in the predictor and therefore, when the SCC unit probes the predictor for speculatively identifying potential data or control invariants, it will provide a prediction based on the current execution state, rather than the state it would be in when the corresponding compacted instruction stream is about to execute post-optimization. This limitation means that when an optimized sequence of micro-ops is streamed, the associated invariants might be identified using histories that are potentially out-of-date. However, as detailed in Section V, our profitability analysis unit ensures that an optimized micro-op sequence is streamed only if the predicted invariants match up with the current state of the predictor, thereby eliminating potential squashing scenarios due to stale predictions.

The code compaction process is considered complete once an appropriate stopping condition is reached. This is similar to the criteria used in Intel processors to decide when to stop streaming micro-ops from micro-op caches. In this particular implementation of SCC, a stopping criteria is satisfied when (a) the end of a 32-byte code region is reached, (b) a micro-op cache miss occurs, or (c) more than two branches are encountered in a 32-byte code region. The

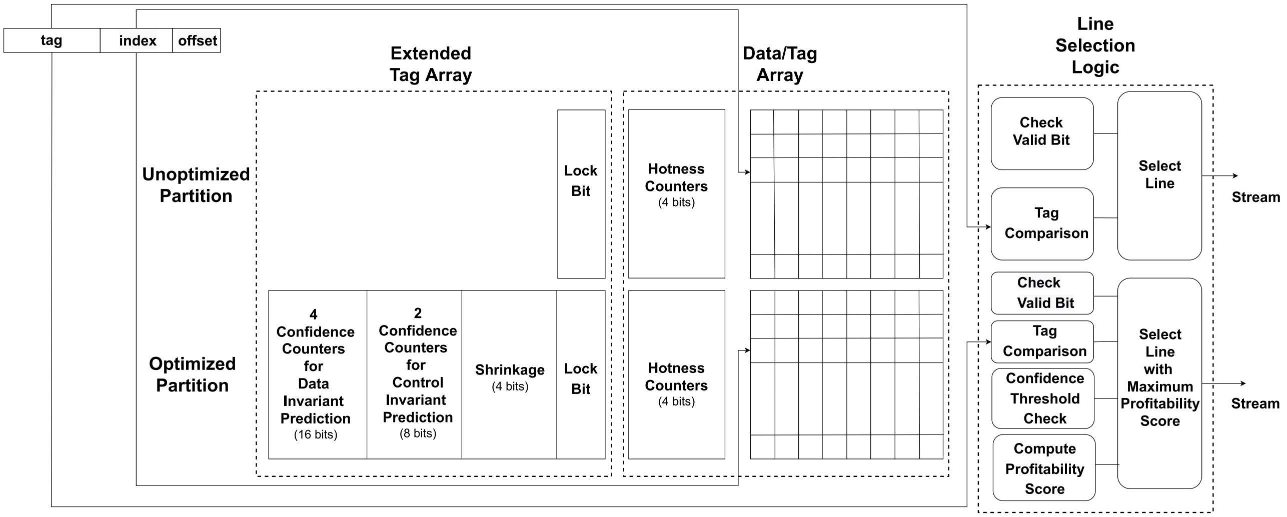

tag
index
offset
Extended Tag Array
Data/Tag Array
Line Selection Logic
Unoptimized Partition
Lock Bit
Hotness Counters (4 bits)
Optimized Partition
4 Confidence Counters for Data Invariant Prediction (16 bits)
2 Confidence Counters for Control Invariant Prediction (8 bits)
Shrinkage (4 bits)
Lock Bit
Hotness Counters (4 bits)
Check Valid Bit
Tag Comparison
Select Line
Stream
Check Valid Bit
Tag Comparison
Confidence Threshold Check
Compute Profitability Score
Select Line with Maximum Profitability Score
Stream

Figure 2: Extensions to the Micro-Op Cache Organization and Line Selection Logic

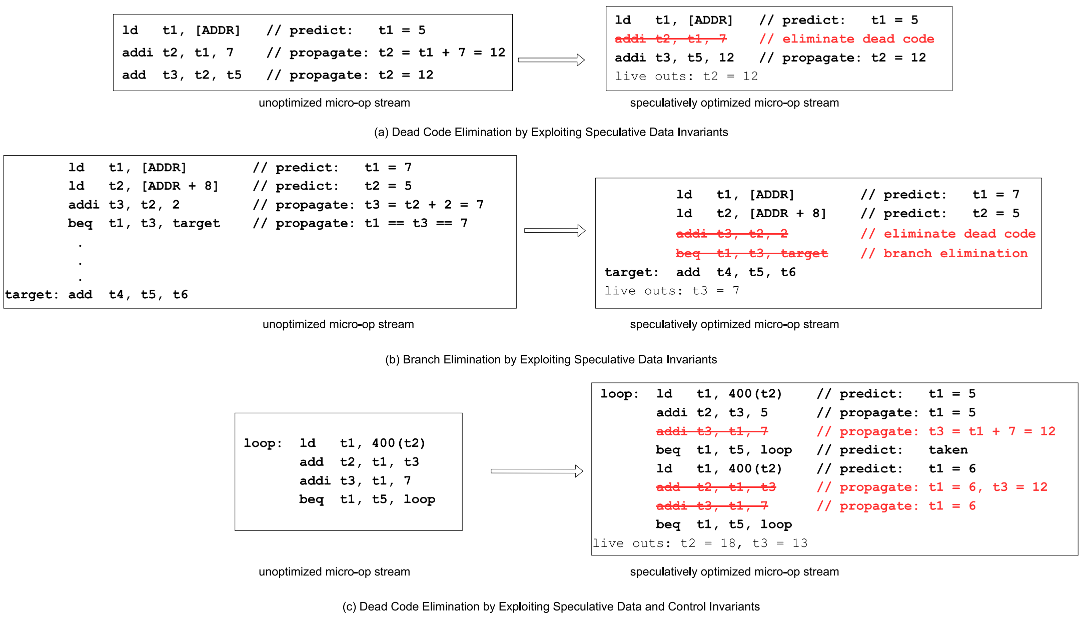
resulting compacted instruction stream in the write buffer is committed to the appropriate location in the micro-op cache, if a predefined compaction threshold is reached, failing which the contents of the write buffer are discarded.

Furthermore, while SCC is able to perform speculative optimizations across basic blocks by identifying control invariants and folding branches, it does not have the ability to do so when presented with self-looping instructions where the macro-operation is broken down into several micro-ops with one of them being a branch micro-op whose target lies within the same macro-operation (a common occurrence in x86-based string manipulation instructions). When such self-looping instructions are encountered or when self-modifying code is detected, the compaction process is considered aborted. However, we note that our ability to detect self-modifying code during speculative code compaction is limited to identifying stores whose addresses: (a) manifest as speculative data invariants, and (b) occur in the same 32-byte code region that is currently being optimized (i.e., same index and tag bits).

Micro-Op Cache Extensions. In accordance with our goal of timely acceleration of sequential code, we extend existing micro-op cache designs to co-host multiple (both unoptimized and optimized) versions of micro-op sequences, to avoid the repeated cost of dynamic code transformation. Similar to multithreaded implementations of Intel micro-op caches [81], [82], [83], we divide the micro-op cache to include separate partitions for unoptimized and optimized micro-op sequences (as shown in Figure 2). The fetch state machine at the front-end is also suitably modified with the ability to dynamically choose from the different versions when available and further stream the appropriate set of micro-ops out of the micro-op cache. In particular, the index bits of the address is used to simultaneously index into both partitions and then tag comparison is performed to identify the appropriate line. However, due to the fact that multiple optimized versions of a given code region may be found in the micro-op cache, it is possible that we produce one hit in the unoptimized partition and multiple hits in the optimized partition. To identify the most profitable instruction sequence to stream, we extend the line selection logic for the optimized partition to only select those lines that meet a dynamically identified confidence threshold (see Section V) and then compute a profitability score (which is essentially a sum of all confidence counters and the compaction potential measured as the shrinkage in the number of instructions) that allows it to select the most profitable instruction stream.

We also extend the tag array of the unoptimized partition to include a lock bit per cache line that will be turned on for those lines currently under code compaction to ensure that they do not get evicted. Note that a maximum of 3 cache lines (i.e., 18 fused micro-ops that belong to the same 32-byte code region) may be locked at any point of time, as that is the granularity of optimization, unlike trace-based architectures that tend to benefit from longer traces. Further, compacted streams that correspond to a 32-byte code region in the optimized partition are tagged by a set of saturating counters to track confidence for each of the predicted invariant (both data and control) in the compacted streams. In our experiments, we find that the best performance benefits are derived through aggressive speculation, and as a result we use 4-bit saturating counters for each of the predicted invariant, allowing us to track a large spectrum of confidence levels. As described in Section V, these counters are updated during instruction execution whenever a prediction is validated.

Fetch Streaming Decisions. Given that the micro-op cache can contain multiple versions of speculatively compacted micro-op sequences, the fetch engine is tasked with the decision of choosing from and streaming the most profitable micro-op sequence into the pipeline, among the various



```
ld   t1, [ADDR]   // predict:   t1 = 5
addi t2, t1, 7    // propagate: t2 = t1 + 7 = 12
add  t3, t2, t5   // propagate: t2 = 12
```

unoptimized micro-op stream

```
ld   t1, [ADDR]   // predict:   t1 = 5
~~addi t2, t1, 7~~    // eliminate dead code
addi t3, t5, 12   // propagate: t2 = 12
live outs: t2 = 12
```

speculatively optimized micro-op stream

(a) Dead Code Elimination by Exploiting Speculative Data Invariants

```
        ld   t1, [ADDR]         // predict:   t1 = 7
        ld   t2, [ADDR + 8]     // predict:   t2 = 5
        addi t3, t2, 2          // propagate: t3 = t2 + 2 = 7
        beq  t1, t3, target     // propagate: t1 == t3 == 7
         .
         .
         .
target: add  t4, t5, t6
```

unoptimized micro-op stream

```
        ld   t1, [ADDR]         // predict:   t1 = 7
        ld   t2, [ADDR + 8]     // predict:   t2 = 5
        ~~addi t3, t2, 2~~          // eliminate dead code
        ~~beq  t1, t3, target~~     // branch elimination
target: add  t4, t5, t6
live outs: t3 = 7
```

speculatively optimized micro-op stream

(b) Branch Elimination by Exploiting Speculative Data Invariants

```
loop:   ld   t1, 400(t2)
        add  t2, t1, t3
        addi t3, t1, 7
        beq  t1, t5, loop
```

unoptimized micro-op stream

```
loop:   ld   t1, 400(t2)     // predict:   t1 = 5
        addi t2, t3, 5       // propagate: t1 = 5
        ~~addi t3, t1, 7~~       // propagate: t3 = t1 + 7 = 12
        beq  t1, t5, loop    // predict:   taken
        ld   t1, 400(t2)     // predict:   t1 = 6
        ~~add  t2, t1, t3~~      // propagate: t1 = 6, t3 = 12
        ~~addi t3, t1, 7~~       // propagate: t1 = 6
        beq  t1, t5, loop
live outs: t2 = 18, t3 = 13
```

speculatively optimized micro-op stream

(c) Dead Code Elimination by Exploiting Speculative Data and Control Invariants

Figure 3: Example illustrating Speculative Code Compaction

choices available. To this end, the fetch engine is equipped with a profitability analysis unit that makes a decision based on three important heuristics – (a) code compaction potential, (b) confidence levels of predicted invariants, and (c) hotness of the compacted code region. It is important to examine all three heuristics in unison since the greatest performance benefits can be derived only by streaming hot code regions that are characterized by both high code compaction potential and low risk of squashing due to mispredictions. The code compaction potential is measured as the number of instructions eliminated by SCC for a particular unoptimized micro-op sequence, and the confidence levels of the predicted invariants are tracked using confidence counters in the extended tag array. The hotness of any given line (in both optimized and unoptimized partitions) is tracked using hotness counters that get incremented every time the line is accessed and decremented periodically. Note that we identify the time period for decreasing hotness through a design exploration that identifies the best threshold (at which the highest performance is observed) for both the baseline and SCC (specifically, every 3 cycles for optimized code and every 28 cycles for unoptimized code).

Misprediction Recovery. Due to the aggressive application of speculative transformations and the fact that the processor front-end typically churns instructions at a much higher rate than the rest of the pipeline, our proposed approach is prone to occasional misspeculation scenarios. Therefore, it is critical that we detect and recover from such scenarios as early in the pipeline as possible, to preserve the performance gains due to speculative code compaction. To this end, we leverage and extend already-existing misprediction recovery mechanisms in modern processors to minimize wasted processor cycles due to misspeculation and further seamlessly redirect control to the appropriate unoptimized micro-op sequence in the micro-op cache when present, to ensure a quick turnaround.

However, as part of the prediction resolution/recovery process, we also update invariant confidences for a compacted instruction stream when a squash or commit signal is received. This allows us to not only reward profitable instruction streams whose predicted data and control invariants continue to hold with high confidence for a long period of time, but also gradually phase out stale ones whose predicted invariants no longer, hold from the micro-op cache.

IV. Speculative Code Compaction

The speculative code compaction process outlined in the previous section entails the application of one of the following speculative transformations on each micro-op being processed. Note that the SCC unit processes micro-ops from the unoptimized partition one at a time, and in program order.

Speculative Data Invariant Identification. Recall that the SCC unit maintains an architectural register file to track potential live values generated along the way. For any micro-op being processed by the SCC unit, its corresponding source operands are first looked up in the register file. If no live values are found, it attempts to speculatively identify a potential invariant by probing the value predictor for the predicted outcome of the micro-op. If a sufficiently high-confidence prediction is available, the predicted outcome is recorded as a speculative data invariant in the SCC register file, so that it can potentially be used while performing speculative transformations such as constant propagation or constant folding on a subsequent dependent micro-op.

If a speculative invariant is identified for a given micro-op, it is marked as a *prediction source* and is retained in the compacted instruction stream to allow us to recover from a potential value misprediction. Note that prediction sources may not be eliminated as they are used for validating predictions. This is illustrated in Figure 3(a), where the load micro-op is speculatively identified as a prediction source and the register *t1* as a speculative data invariant. We find that redundant loads that access hot data structures, such as large matrices that are infrequently updated, but frequently accessed, are one of the prime sources of speculative data invariants, in addition to highly predictable program counter-relative loads that access data from constant pools.

Speculative Constant Folding. If the values of all the source operands of the micro-op being processed are available and valid in the SCC register file (i.e., they were identified as speculative invariants by an older micro-op in the unoptimized instruction stream being processed), they are fed as live input values to the SCC ALU, which then speculatively evaluates the micro-op as a constant expression, and subsequently eliminates it as dead code. Following our example in Figure 3(a), the *addi* instruction has one of its source operands *register t1* available in the SCC register file as a live value (previously identified as a speculative invariant). Since its other source operand is an immediate value, the entire operation can be speculatively folded away, eliminating the entire micro-op. The destination register *t2* is recorded as a *live out* in the SCC register file with its speculatively folded value (in this case, 12).

Speculative Constant Propagation. If the values of only some source operands are available in the SCC register file, an addressing mode transformation is applied such that the micro-op is converted to its corresponding immediate version. That is, speculative data invariants that manifest as constants are encoded in the immediate fields of the micro-op. This entails editing appropriate fields in the decoded micro-op to ensure that the respective source operands are immediately available as constants rather than requiring a register lookup. In our example from Figure 3(a), this transformation is applied on the *add* micro-op, since its source operand *t5* is not available as a live value in the SCC register file, although its other source operand *t2* is. Note that constant propagation reduces the number of renaming operations and register lookups, while simultaneously improving the effective instruction-level parallelism, due to fewer overall read-after-write dependencies.

Speculative Branch Folding. If the micro-op is a control instruction, but the branch direction and target can be deduced based on the speculatively identified live values in the SCC register file, the entire branch can be eliminated. Further, the SCC unit is then instructed to pivot and start processing micro-ops at the speculatively identified branch target, in the next cycle, provided that they are resident in the micro-op cache. This is illustrated in our example from Figure 3(b), where the *beq* instruction gets eliminated since the branch direction and target can be evaluated based on the speculatively identified live values of its source registers *t1* and *t3*. In this particular case, the branch direction speculatively evaluates to *taken*, and the SCC unit is instructed to pivot to the *add* micro-op that can be found at the branch *target*.

Speculative Control Invariant Identification. Similar to the case of speculative data-invariant identification, the SCC unit may also probe the branch prediction unit to identify potential control invariants, in case the micro-op being processed is a branch operation that cannot be folded. Again, if a high-confidence prediction is available, the instruction is marked as a *prediction source* and therefore may not be eliminated). Just like the case of speculative branch folding, the predicted target is then used in the next cycle to obtain the next micro-op to be processed, as long as it is resident in the micro-op cache. This can be observed from our example in Figure 3(c), where the *beq* instruction is speculatively identified as a prediction source, and is predicted by the branch predictor to be *taken* with high confidence. As a result, the SCC unit starts processing micro-ops at the predicted target (i.e., *loop*) in the next cycle. This allows us speculatively identify values that manifest as invariants across different basic blocks, uncovering greater potential for dead code elimination, as shown in the example.

Inlining Live Outs. Finally, for every micro-op that is a *prediction source* (both data and control sources), all speculatively identified live data values in the SCC register file, excluding the one generated by that micro-op, are marked as live outs, to be made visible at the time of instruction rename, similar to existing rename-time optimizations such as move elimination. This ensures that the register state remains consistent in the event of a potential value misprediction. Live outs are also inlined at the end of every compacted instruction stream (i.e., when the last micro-op of a compacted instruction stream is issued), so that they become immediately available to subsequent younger instructions in the pipeline. To perform this efficiently, we leverage existing rename-time schemes such as Physical Register Inlining [84] that has the ability to efficiently track constants of narrow widths in renaming structures, thereby limiting the impact on access latency and hardware overhead, and then propagate them to dependent instructions. We include a sensitivity analysis in Section VII to evaluate the feasibility of leveraging such techniques.

To illustrate the full process of speculative code compaction for a given hot code region resident in the micro-op cache, consider Figure 4, which shows (a) a compiler-optimized basic block in the SPEC CPU 2017 benchmark, *xalancbmk*, that consists of 7 native x86 instructions, (b) its corresponding

```
I1:   je      497740
I2:   mov     (%r15), %rax
I3:   test    %rax,   %rax
I4:   je      4977f1
I5:   mov     (%rax), %eax
I6:   test    %eax,   %eax
I7:   je      4977f1
```

(a) x86 Assembly Code Stream

```
I1-1:  rdip   t1,   %ctrl153
I1-2:  limm   t2,   ffffffde
I1-3:  wrip   t1,   t2
I2-1:  ld     rax,  (r15)
I3-1:  test   eax,  eax
I4-1:  rdip   t1,   %ctrl153
I4-2:  limm   t2,   ffffffde
I4-3:  wrip   t1,   t2
I5-1:  ld     eax,  (rax)
I6-1:  test   eax,  eax
I7-1:  rdip   t1,   %ctrl153
I7-2:  limm   t2,   79
I7-3:  wrip   t1,   t2
```

(b) Unoptimized Micro-op Stream

```
I1-3:  wrip   t1,   t2       // predict: t1  = 0x497782
I2-1:  ld     rax,  (r15)    // predict: rax = 0xac74d0
I5-1:  ld     eax,  (rax)    // predict: eax = 0x1
I7-3:  wrip   t1,   t2       // predict: t1  = 0x49778
```

(c) Speculatively Optimized Micro-op Stream

Figure 4: Speculative Code Compaction on a hot kernel found in the SPEC CPU 2017 application *xalancbmk*

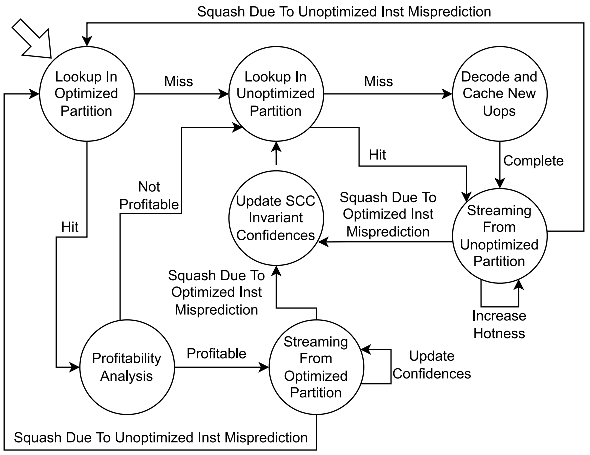

Squash Due To Unoptimized Inst Misprediction
Lookup In Optimized Partition
Miss
Lookup In Unoptimized Partition
Miss
Decode and Cache New Uops
Complete
Hit
Hit
Not Profitable
Update SCC Invariant Confidences
Squash Due To Optimized Inst Misprediction
Streaming From Unoptimized Partition
Increase Hotness
Squash Due To Optimized Inst Misprediction
Profitability Analysis
Profitable
Streaming From Optimized Partition
Update Confidences
Squash Due To Unoptimized Inst Misprediction

Figure 5: Fetch State Machine extended for SCC

micro-op translation, which consists of 13 micro-ops, and (c) the compacted version of the given unoptimized micro-op sequence, which consists of just 4 micro-ops, completely folding away the branch, redirecting execution to the appropriate branch target.

V. The Fetch State Machine

In this section, we describe extensions to the fetch unit that allow us to make profitable streaming decisions and gracefully recover from misspeculation, thereby maximizing the gains due to speculative code compaction.

Figure 5 shows the modified state machine for an x86-based fetch engine that already has the ability to switch back and forth between the micro-op cache and the traditional decode pipeline. Note that SCC introduces an additional source for fetching micro-ops, i.e., the optimized micro-op cache partition, and in ensuring that micro-op sequences are sourced profitably from the appropriate partitions, it introduces a few additional transitions to the fetch state machine, the rationale for which is described below.

Profitability Analysis. Given the inherently speculative nature of the proposed transformations, it is imperative that the fetch engine selectively optimizes only those micro-op sequences where the benefits of SCC outweigh the potential squashing overheads. To this end, we modify the fetch engine such that it is able to (a) dynamically identify and reward well-behaving speculatively-optimized instruction streams that are characterized by high compaction but incur few mispredictions if any, and (b) penalize and eventually phase out instruction streams whose predicted invariants have become stale over time, not only limiting squashing, but also making room in the micro-op cache for newer and potentially more useful instruction streams.

To identify well-behaving and profitable instruction streams, the fetch engine is equipped with a profitability analysis unit that is triggered when multiple versions of a micro-op sequence (both unoptimized and speculatively optimized) are available in the micro-op cache for streaming. The profitability analysis unit determines that the speculatively-optimized instruction sequence is indeed profitable for streaming if the following conditions are met: first, the saturating confidence counters for the speculatively identified control invariants indicate that the instruction stream has not crossed a dynamically identified threshold of mispredictions (that is tuned on the basis of the rate at which mispredictions increase or decrease); second, all of the predicted data invariants match up with the current state of the value predictor, indicating that the invariants identified at the time of optimization continue to hold; third, the instruction stream is characterized by high compaction potential in terms of the number of micro-ops deemed speculatively dead and eliminated; and finally, the instruction stream meets a predefined hotness threshold.

Further, when multiple speculatively-optimized instruction streams are available due to difference in the number, range, and confidence of predicted invariants used for optimization, the instruction stream that has the highest data invariant confidence and provides the greatest compaction is chosen. Hosting and streaming from multiple speculatively-optimized instruction streams is a unique feature of this work, and we observe that this is especially useful for code regions where

Baseline Processor			
Frequency	2.4 GHz	ICache	32 KB, 8-way
Fetch width	6 fused μops	DCache	48 KB, 8-way
μop cache	2304 μops, 8-way	IDQ	140 entries
Branch Predictor	LTAGE	Value Predictor	H3VP
RAS	64 entries	BTB	4096 entries
Dispatch width	10 unfused μops	ROB	352 entries
Register file	256 INT/FP	LQ/SQ	128/72 entries
Commit width	10 unfused μops	Functional	Int ALU (6)
Squash width	10 unfused μops	Units	FPALU (3)
L2 Cache	512 KB 8-way	L3 Cache	8 MB 16-way
L1I/L1D Repl. Pol.	LRU	L2/L3 Repl. Pol.	Random

Table I: Microarchitectural Configuration Parameters

the outcome of an instruction oscillates predictably, with high confidence, albeit between a limited set of data values.

Misspeculation Recovery. To limit the squashing overheads introduced by a few select instruction streams that feature aggressive speculation, we extend the misprediction recovery mechanism to ensure that, regardless of the decision made by the profitability analysis unit, we automatically select the unoptimized micro-op cache partition as our fetch source when the following conditions are met (as shown in Figure 5): first, the offending instruction that caused the misprediction was issued out of the speculatively-optimized micro-op cache partition and was marked as a valid prediction source by the SCC unit; second, the reason for misspeculation is due to a speculative processor feature that is related to the optimizations enabled by SCC (for example, it is not due to speculative memory disambiguation). These conditions are instrumental in ensuring that we stop fetching stale instruction streams from the optimized partition whose predicted invariants no longer continue to hold, and as a result get gradually phased out and replaced by newer, more relevant, and hot instruction streams.

Finally, we also ensure that confidence counters in speculation structures such as the branch and value predictors are always updated, even if the squash/commit signal is received due to a speculatively optimized instruction. This is important because it ensures that predictor state always remains current and does not go out-of-sync when speculatively-optimized instruction streams are being executed.

VI. Methodology

This section describes our baseline architecture, modeling assumptions, and experimental methodology.

Baseline and Modeling Assumptions. Table I describes the microarchitectural configuration of our baseline processor, which is modeled after Intel's Icelake microarchitecture. We use the gem5 [85] architectural simulator for performance evaluation and the McPAT [80] framework for modeling area and power overheads due to the additional structures introduced in SCC. We integrate the H3VP [86] and EVES [87] predictors from the 2019 Championship Value Prediction (CVP) into our simulator framework, as we use value prediction as our primary mechanism to predict data invariants. Finally, we also extend both gem5 and McPAT to model micro-op caches and micro-op fusion as described in Intel's Architecture Optimization Manual [81], with a hotness-based replacement policy as described by Ren, et al. [82]. While these extensions allow us to model Intel's front-end microarchitecture as closely as possible, on par with or ahead of contemporary research in the field, certain simulator-imposed limitations still exist. In particular, due to the propreitary nature of the macro-to-micro-op mapping used by Intel, we rely on the mapping scheme provided by the gem5 simulator, which might not necessarily capture potential optimized mapping schemes used in state-of-the-art commercial implementations.

Workload Selection. To evaluate the performance and energy savings potential of SCC, we construct workloads using 11 benchmarks from the SPEC CPU 2017 benchmark suite [88] and 8 benchmarks from the PARSEC 3.0 benchmark suite. We include all SPEC CPU 2017 INT applications except x264 and omnetpp. We were unable to successfully run x264 on gem5 (even on the latest version without our changes) due to a bug in the implementation of certain x86 SSE instructions that the benchmark uses. However, we include the PARSEC version of the benchmark x264 in our analysis. Furthermore, our checkpoint generation simulations for omnetpp failed due to high memory usage, as a result of which we were able to generate a checkpoint for one only simpoint (with a weight of 0.16). While we observe a speedup of 48% for this simpoint, we did not include it in the paper as it is not representative of the entire application and could potentially artificially inflate our speedup.

To account for phase behavior, we use the SimPoint [89] methodology to break down each application into multiple simpoints that include representative runs of 100 million dynamic instruction intervals sampled using the PIN instrumentation framework [90]. Since our goal is to speculatively optimize already compile-time optimized code, we compile all benchmarks with the highest level of optimization using the LLVM compiler framework [91].

VII. Results

In this section, we first discuss results from our best microarchitectural configurations and then delve into detailed sensitivity studies that show the effect of various parameters and microarchitectural structures on our ability to successfully optimize inherently sequential code.

A. Performance Evaluation

Code Compaction Potential. Figures 6 (top panel) shows the compaction potential of SCC (broken down by each individual optimization) measured in terms of the dynamic committed instruction count reduction achieved on already compile-time optimized code. Note that we also include the partitioned baseline (although it performs similarly to the original baseline) as it is an important step in enabling SCC.

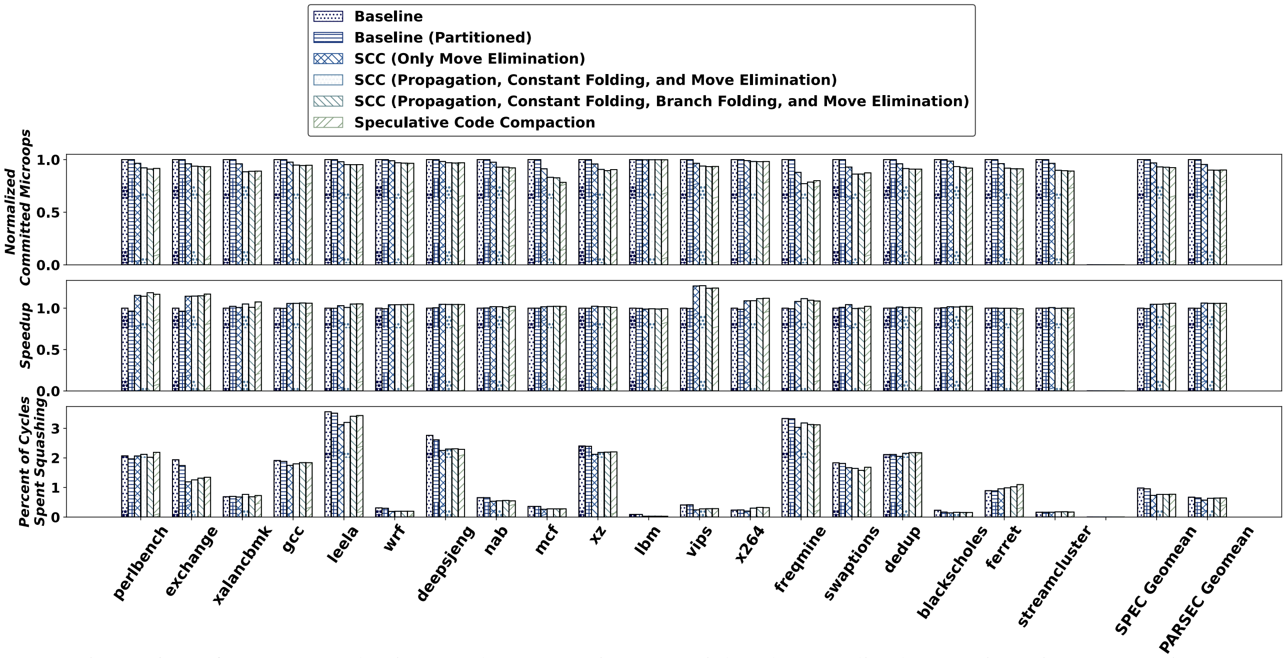


Figure 6: Performance Evaluation: Code Compaction Potential and Normalized Execution Time Comparison

We make several key observations from this experiment. First, we find that the majority of our code compaction benefits arise from applying speculative optimizations on short micro-op sequences that occur within a basic block. This includes identifying speculative data invariants and then performing constant folding and propagation (fourth bar from the left). Second, we observe that a considerable chunk of the micro-ops eliminated via constant folding are comprised of *register-immediate* move instructions (third bar from the left), calling for more aggressive move and zero/one idiom elimination at the front-end of the pipeline, even before the rename stage. Third, although we observe that speculative data invariants get routinely propagated as constants to dependent instructions across branch boundaries, the potential for speculative folding constants and thereby eliminating instructions across basic blocks seems to be limited. While more code compaction is possible through the aggressive application of speculative optimizations across different basic blocks, such instruction streams are also more prone to squashing and are deemed less profitable for streaming.

Overall, we see uniformly high instruction count reduction on all applications, except the benchmarks *lbm*, *wrf*, and *x264* that spend most of their time executing floating-point and SIMD instructions that are currently unoptimizable by SCC. On average, we are able achieve an instruction count reduction of 7.63% for SPEC and 9.86% for PARSEC.

While the goal of SCC is to eliminate wasteful computation and generate compact instruction streams, it also does so speculatively and therefore runs the risk of incurring more than usual squashing overheads. As a result, dynamic instruction count as a metric, while useful to understand the extent of achievable code compaction, does not provide sufficient insights into the overall profitability of the technique.

Impact on Execution Time. To examine its performance potential, we next turn to Figure 6 (middle panel) that compares SCC with the baseline architecture, in terms of the normalized execution time, again broken down by different optimizations. We notice several interesting trends and make high-level observations about the different benchmarks. First, workloads with high data and control predictability such as *freqmine*, *perlbench*, and *xalancbmk*, benefit the most from SCC. In these applications, we observe that as the level of optimization increases, there is also a small uptick in the squashing overhead (as seen from the bottom panel of Figure 6 that counteracts the benefit due code compaction. Second, in benchmarks such as *vips* and *exchange*, we observe a significant speedup due to speculative move elimination alone, despite its limited ability to reduce the dynamic instruction count on those benchmarks. Upon further analysis, we are able to uncover an interesting side-effect of SCC that significantly improves the predictability of certain hard-to-predict branches. More specifically, we observe that the top ten critical branches in these applications occur as part of compacted instruction streams that are deemed profitable, reducing their misprediction rate over the course of execution. We attribute this to the fact that the confidence counters associated with each compacted instruction stream (stored as part of the extended tag array in the micro-op cache) also act as local predictors for branches occurring in those

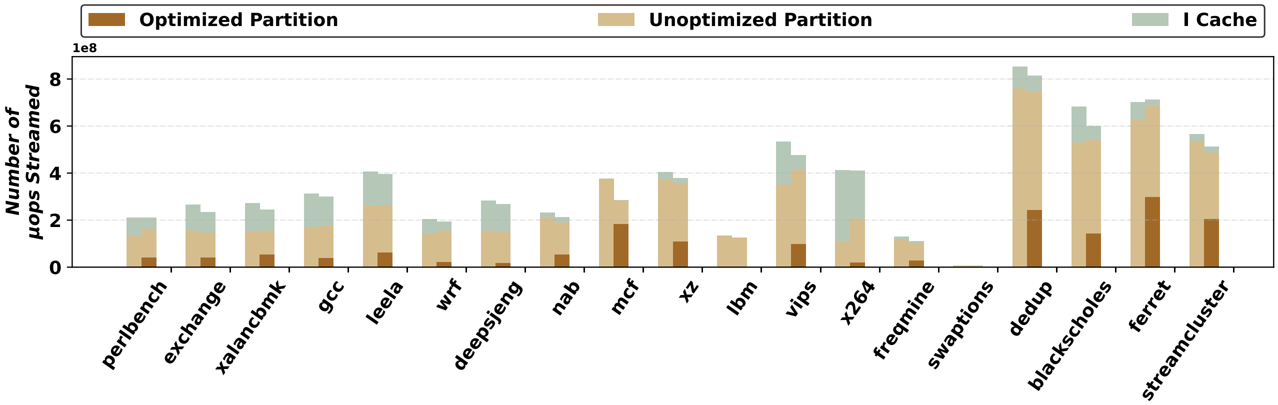


Figure 7: Examining Fetch Sources to analyze the impact on Micro-Op Cache Utilization

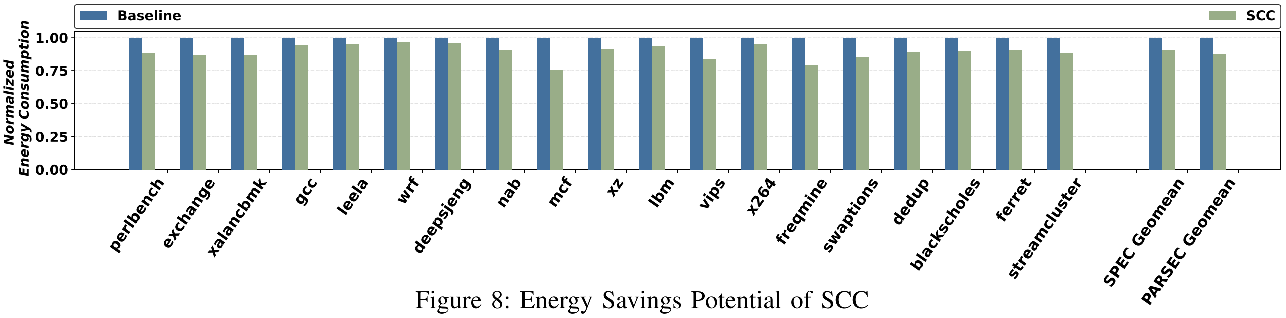


Figure 8: Energy Savings Potential of SCC

streams, providing high confidence predictions and in many cases, limiting the impact of negative branch interference. Third, we observe that inherently memory-bound applications such as *mcf* and *xz* do not benefit from SCC from a performance standpoint, despite their potential for high instruction count reduction. Similarly, we do not observe a speedup on low ILP application such as *leela* and *swaptions* due to frequent reorder buffer full scenarios. In contrast, on high ILP applications such as *deepsjeng* and *streamcluster*, we observe a limited speedup as they are bottlenecked by a finite instruction queue.

Overall, we outperform the baseline architecture by an average of 5.81% for SPEC and 5.80% for PARSEC.

Impact on Micro-Op Cache Utilization. One of the major features of SCC is its ability to co-host unoptimized and optimized micro-op sequences in the micro-op cache, and allow the fetch engine to choose from and stream different versions of unoptimized and optimized micro-op sequences based on the profitability heuristics. Figure 7 shows the number of micro-ops sourced by the fetch engine from: (a) the instruction cache, (b) the unoptimized micro-op cache partition, and (c) the optimized micro-op cache partition, for both SCC (right bar) and the baseline (left bar). We draw two major conclusions from this result. First, the number of instruction cache accesses for SCC is consistently lower than that of the baseline across most benchmarks, indicating that the micro-op cache utilization improved, despite it being partitioned amongst unoptimized and optimized instruction sequences. This is because partitioning the micro-op cache has limited impact on performance (as shown in Figure 6) due to the already high number of conflict misses in certain critical cache sets, but more importantly, top hot code regions in the micro-op cache tend to get optimized by SCC, and if deemed profitable, tend to stay in the optimized partition, freeing up space in the micro-op cache, resulting in fewer micro-op cache misses and saving the relatively expensive trip (in terms of both time and energy) to the instruction cache. In other words, organizing the micro-op cache into optimized and unoptimized partitions also has the side-effect of improving the associativity of certain critical code regions that map to the same set. This phenomenon can be evidenced in the application *x264* where the micro-op cache hit rate gets almost doubled due to fewer conflicts in certain critical sets. Second, for most benchmarks, the fraction of micro-ops streamed from the optimized partition dominates the overall fetch bandwidth. This implies that not only do we have the ability to aggressively optimize hot code regions in the micro-op cache, but our profitability heuristics allow us to successfully ensure that the most profitable compacted instruction streams stay in the optimized partition.

B. Power, Area, and Energy Evaluation

We next turn our attention to the energy savings potential of SCC (considering the entire chip). Clearly, from Figure 8, we see that the energy savings benefits of SCC is even greater than its performance potential. In fact, even though

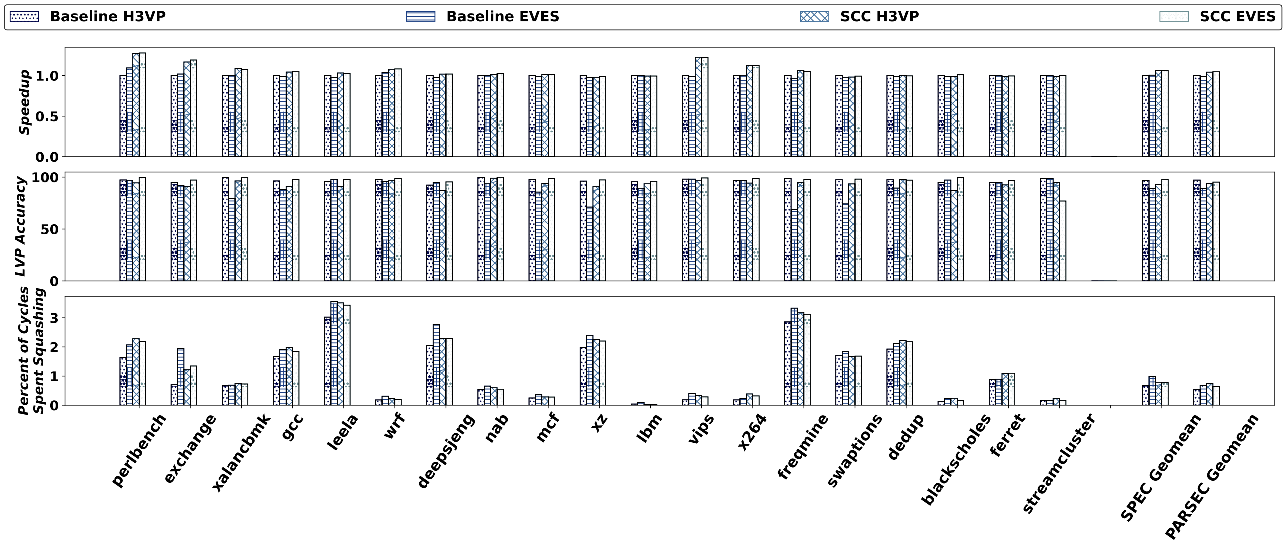


Figure 9: Sensitivity to different Value Predictor Designs: EVES vs. H3VP

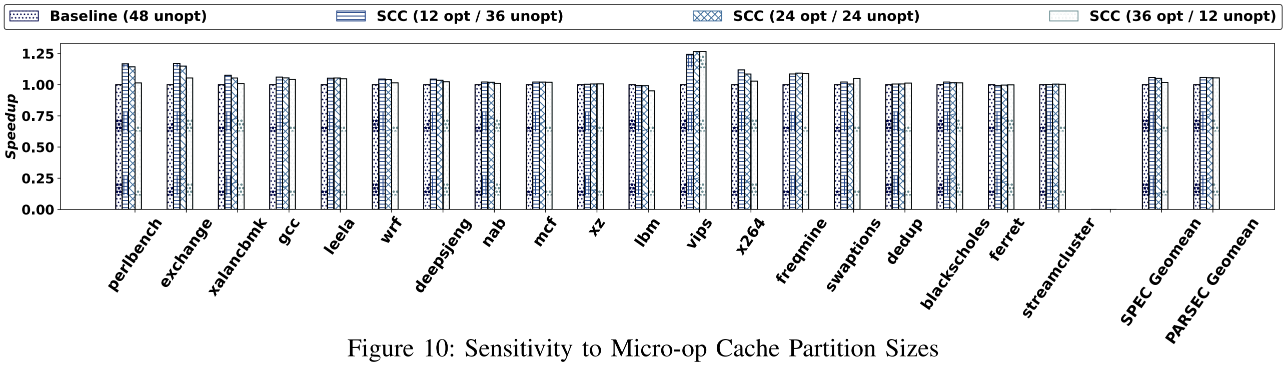


Figure 10: Sensitivity to Micro-op Cache Partition Sizes

we observe only a small speedup in benchmarks such as *xz* and *mcf*, we consistently save more than 20% in energy for most applications, going up to 24% for SPEC and 22% for PARSEC. This is not only due to the elimination of dead code propagating through much of the processor's back end, but also because we have greatly improved the hit rate and average residency in the micro-op cache, by dividing it into optimized and unoptimized partitions, converting several misses into hits, significantly reducing the energy expended while making instruction cache accesses.

Furthermore, recall that SCC is a minimally-invasive technique that only adds a small integer ALU and a register file to keep track of speculatively identified live values; the resulting area and peak power overheads of these minor extensions to the front-end are a meagre 1.5% and 0.62% respectively. Overall, we achieve an average of 12% savings in energy consumption.

C. Sensitivity Analysis

Value Predictor Configurations. We explore two state-of-the-art value predictors from the Championship Value Prediction workshop: (a) H3VP, which is a 3-period predictor that captures oscillating patterns, and (b) EVES, which leverages enhanced stride history. Figure 9 shows the result of this exploration. We observe that both predictors perform similarly across most benchmarks in the specific context of SCC, despite the fact that EVES is known to provide higher accuracy and avoid penalties due to squashing, as can be seen from the middle and bottom panels of the figure. Notable exceptions include *xalancbmk*, where H3VP outperforms EVES, suggesting that the application benefits from aggressive speculative optimizations. In this particular case, we observe that our profitability analysis unit already takes care of penalizing and phasing out speculatively optimized instruction streams with high misprediction rate. However, this depends highly on the nature of the workload. For example, on applications such as *gcc*, we observe that EVES provides better performance with SCC by avoiding expensive squash penalties, while simultaneously identifying a tangible set of instruction sequences that can benefit from speculative code compaction.

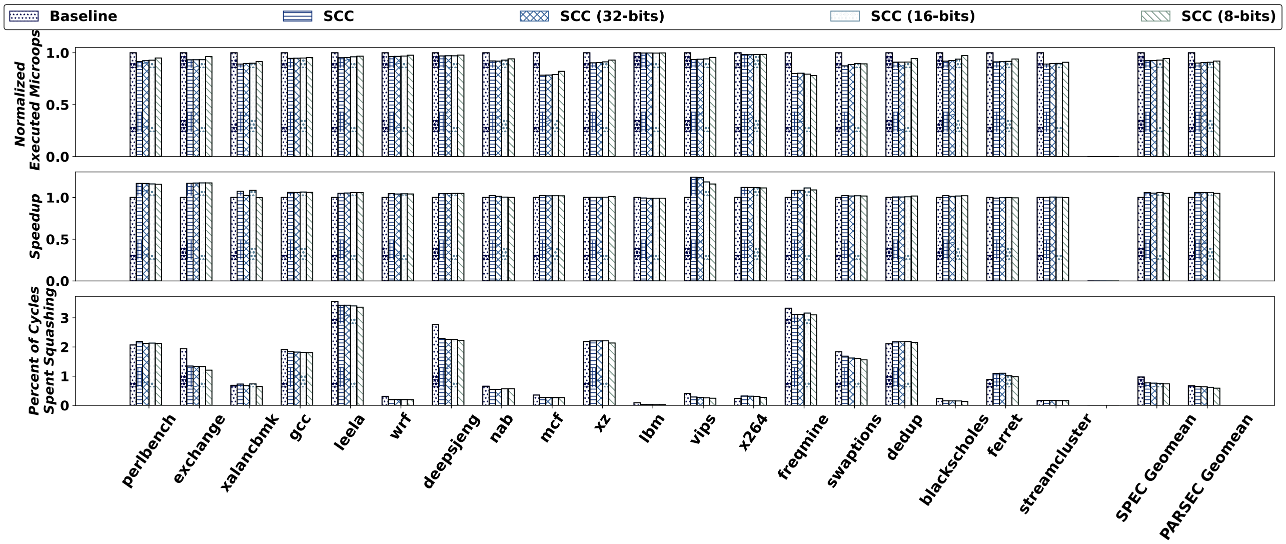


Figure 11: Sensitivity to Constant Width Configurations

Micro-op Cache Partition Sizes. We next examine the sensitivity of SCC to the micro-op cache partition sizes. We explore three configurations where we allocate – (a) one-third (12 sets), (b) half (24 sets), and (c) two-thirds (36 sets), to the optimized partition of the micro-op cache (as shown in Figure 10). We find that best benefits are obtained when the unoptimized partition is larger than the optimized partition (i.e., 12 sets and 36 sets respectively). We attribute this to the following major reasons. First, a larger unoptimized partition allows us to track a greater chunk of hot code regions and thereby facilitate greater code compaction. Second, the optimized partition typically also exhibits greater utilization due to the fact it stores compacted instruction streams with high shrinkage that have been deemed profitable over the course of the application's execution.

Constant Widths. Finally, recall that, for every prediction source, speculatively computed live-outs need to be made available to dependent instructions in their *immediate* form, to ensure a consistent register state in the event of a potential misprediction. To examine the feasibility of efficiently performing this by leveraging a low-overhead rename-time approach such as physical register inlining [84], we explore multiple constant widths. Figure 11 shows the result of this experiment. We observe that we are able to retain most of our benefits even when we are restricted to only propagate constants that are 16-bit wide, and a further restriction to propagating only 8-bit constants impacts instruction count reduction by 6.8% and slows down performance by only 4.9% on average in comparison to no restriction on the constant widths. Furthermore, we observe only 0.78% of the dynamic instructions (averaged across all benchmarks) carry live-outs over the course of execution, with 0.62% carrying only one live-out and 0.11% carrying two live-outs. This effectively allows us to propagate live-out values in rename-time structures such as the map table without significant impact on latency and hardware overhead.

VIII. Conclusion

This work proposes Speculative Code Compaction (SCC), an aggressive scheme of dynamic binary optimizations, implemented entirely in hardware within the processor, while incurring just 1.5% in area overhead. It speculatively compacts hot code regions in the micro-op cache by leveraging a wide array of existing speculation techniques in modern processors, such as branch prediction and value prediction. Due to the aggressive deployment of speculative transformations, SCC outperforms a baseline processor architecture modeled after Intel's Icelake by as much as 18% for SPEC and 30% for PARSEC, providing speedups on traditionally low ILP applications, and saves as much as 22% in energy for PARSEC and 24% for SPEC.

Acknowledgments

The authors would like to thank the anonymous reviewers and the shepherd for their many helpful suggestions and comments. This research was supported by NSF Grant CNS-1850436, NSF/Intel Foundational Microarchitecture Research (FoMR) Grant CCF-1912608, a Semiconductor Research Corporation (SRC) contract 2019-NM-2875, and CRISP, one of six centers in JUMP, an SRC program, sponsored by MARCO and DARPA.

Appendix

A. Abstract

The artifact for this paper describes the frameworks used for our evaluations. It consists of a simulation infrastructure that evaluates the performance and power usage of SCC. The main results of the paper from Figure 6 can be reproduced using the gem5 simulator. The power results in Figure 8 can be reproduced using McPAT. Specifically, we provide scripts to reproduce results for:

- **Figure 6 (Top):** Executed Micro-op Count for both *Baseline* and *SCC*
- **Figure 6 (Middle):** Speedup for both *Baseline* and *SCC*
- **Figure 8:** Energy Consumption for both *Baseline* and *SCC*

Please note that the scripts can be modified to reproduce any of the results presented in the paper, including the sensitivity experiments.

B. *Artifact check-list (meta-information)*

- **Program:** SPEC CPU2017 and PARSEC 3.0
- **Compilation:** LLVM with -O3
- **Run-time environment:** All simulations were preformed on CentOS Linux release 7.9.2009
- **Hardware:** Gem5 simulator
- **Metrics:** Execution time, instruction count reduction, and dynamic energy consumption
- **Output:** The gem5 simulations provides a statistics output file, containing cycles spent and overall dynamic instruction count. McPAT provides detailed power consumption characteristics.
- **Experiments:** Scripts and instructions are provided in the artifact README files.
- **How much disk space required (approximately)?:** About 250 GB of disk space is required for the SPEC 2017 simpoints, and around 5 GB is needed for the gem5 code and binaries.
- **How much time is needed to prepare workflow (approximately)?:** About 30 minutes to download the frameworks and install requirements, and around 30 minutes to compile gem5.
- **How much time is needed to complete experiments (approximately)?:** Assuming enough available parallelism, the gem5 experiments need at least 6 hours.
- **Publicly available?:** Yes, the code is available on Github (see Section A.3.1).
- **Code licenses (if publicly available)?:** GPL v3
- **Workflow framework used?:** Yes, bash and slurm scripts are provided.
- **Archived:** 10.5281/zenodo.7018601

C. *Description*

1) How to access: The artifact is available on github at the following URL: https://github.com/logangregorym/gem5-changes.

2) Hardware dependencies: Any modern Linux cluster should be able to reproduce these experimental results.

3) Software dependencies: We provide Simpoints created from SPEC 2017 benchmarks for the artifact evaluators, but we cannot publish them as they are under copyright. Other programs used for evaluations are publicly available.

D. *Installation*

The artifact provides scripts to install requirements as well as building the provided tools.

E. *Experiment workflow*

This section provides a high-level overview of the experimental workflow. Please follow the instructions in the README for a detailed, step-by-step guide.

For the performance evaluation, we need to first compile the gem5 code and prepare our benchmark programs. Then, we run multiple gem5 simulations for each set of benchmark programs. Each experiment is configured to represent one of the following optimization levels: (1) a baseline with classic value prediction, (2) the same baseline with half the micro-op cache size, (3) SCC with simple move elimination, (4) SCC with propagation, constant folding, and move elimination, (5) SCC with propagation, constant folding, branch folding, and move elimination, and (6) full Speculative Code Compaction. For artifact evaluation we provide scripts for reproducing the main result, (1) and (6). However, the scripts can be easily modified to produce any of the listed experiments. Finally, once the simulations are completed, we run the provided scripts to extract the results from gem5 simulations and plot the figures.

To obtain power results, we run McPAT on the completed gem5 simulation stats files to obtain dynamic energy consumption numbers. Then a script to extract the numbers and plot the graphs is provided.

F. *Evaluation and expected results*

The gem5 simulations should result in performance, instruction count, and energy numbers that match exactly those presented in the paper.

G. *Experiment customization*

All gem5 options can be viewed by running the command:

```
./build/X86/gem5.opt ./configs/example/se.py \\
--help
```

The options for running baseline simulations are:

```
--caches --l2cache --cpu-type=O3_X86_icelake_1 \\
--mem-type=DDR4_2400_16x4 --mem-size=64GB \\
--mem-channels=2  --enable-microop-cache \\
--enable-micro-fusion --lvpredType=eves \\
--predictionConfidenceThreshold=15  --l3cache\\
--enableValuePredForwarding \\
--predictingArithmetic=1 \\
--enableDynamicThreshold --forceNoTSO \\
--uopCacheNumSets=48 --uopCacheNumWays=8 \\
--uopCacheNumUops=6
```

The options for running SCC simulations are:

```
--caches --l2cache --cpu-type=O3_X86_icelake_1 \\
--mem-type=DDR4_2400_16x4 --mem-size=64GB \\
--mem-channels=2  --enable-microop-cache \\
--enable-micro-fusion --lvpredType=eves \\
--enable-superoptimization \\
--predictingArithmetic=1 \\
--usingControlTracking=1 --usingCCTracking=1 \\
--predictionConfidenceThreshold=5 \\
--uopCacheNumSets=24 --uopCacheNumWays=8 \\
--uopCacheNumUops=6 --specCacheNumWays=4 \\
--specCacheNumSets=24 --specCacheNumUops=6 \\
--l3cache --lvpLookupAtFetch \\
--enableDynamicThreshold --forceNoTSO
```

H. Methodology

Submission, reviewing, and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] L. A. Barroso and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition," *Synthesis Lectures on Computer Architecture*, Jul 2013.

[2] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing," in *Proceedings of the VLDB Endowment*, Dec 2017.

[3] S. D. Fihn, J. Francis, C. Clancy, C. Nielson, K. Nelson, J. Rumsfeld, T. Cullen, J. Bates, and G. L. Graham, "Insights from Advanced Analytics at the Veterans Health Administration," *Health Affairs*, Jul 2014.

[4] R. Cheng, J. Hong, A. Kyrola, Y. Miao, X. Weng, M. Wu, F. Yang, L. Zhou, F. Zhao, and E. Chen, "Kineograph: Taking the Pulse of a Fast-Changing and Connected World," in *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[5] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2003.

[6] R. Joshi, M. D. Bond, and C. Zilles, "Targeted Path Profiling: Lower Overhead Path Profiling for Staged Dynamic Optimization Systems," in *Proceedings of the 2nd Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2004.

[7] S. Hu and J. E. Smith, "Using Dynamic Binary Translation to Fuse Dependent Instructions," in *Proceedings of the 2nd Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2004.

[8] M. D. Bond and K. S. McKinley, "Practical Path Profiling for Dynamic Optimizers," in *Proceedings of the 3rd IEEE/ACM International Symposium on Code Generation and Optimization*, 2005.

[9] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[10] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *ACM SIGPLAN Notices*, Jun 2005.

[11] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[12] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," in *Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization*, 2003.

[13] M. DeVuyst, A. Venkat, and D. M. Tullsen, "Execution migration in a heterogeneous-isa chip multiprocessor," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[14] A. Venkat and D. M. Tullsen, "Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor," in *Proceedings of the International Symposium on Computer Architecture*, 2014.

[15] A. Venkat, A. Krishnaswamy, K. Yamada, and R. Palanivel, "Binary Translation driven Program State Relocation," in *United States Patent Grant US009135435B2*, 2015.

[16] A. Venkat, S. Shamasunder, H. Shacham, and D. M. Tullsen, "Hipstr: Heterogeneous-isa program state relocation," in *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems*, 2016.

[17] A. Venkat, *Breaking the ISA Barrier in Modern Computing*. PhD thesis, UC San Diego, 2018.

[18] A. Venkat and T. M. D. Basavaraj, Harsha, "Composite-ISA Cores: Enabling Multi=ISA Heterogeneity using a Single ISA," in *HPCA*, 2019.

[19] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. Jun 2005.

[20] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient Flow Tracing with Dynamic Binary Rewriting," in *Proceedings of the 11th IEEE Symposium on Computers and Communications*, 2006.

[21] J. Yang, K. Skadron, M. L. Soffa, and K. Whitehouse, "Feasibility of Dynamic Binary Parallelization," in *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, May 2011.

[22] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-Guided Automated Software Diversity," in *Proceedings of the 11th IEEE/ACM International Symposium on Code Generation and Optimization*, 2013.

[23] H. Massalin, "Superoptimizer: A Look at the Smallest Program," in *ACM SIGPLAN Notices*, Oct 1987.

[24] S. Bansal and A. Aiken, "Automatic Generation of Peephole Superoptimizers," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2006.

[25] S. Bansal and A. Aiken, "Binary Translation using Peephole Superoptimizers," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, Dec 2008.

[26] T. Granlund and R. Kenner, "Eliminating Branches Using a Superoptimizer and the GNU C Compiler," *ACM SIGPLAN Notices*, Jul 1992.

[27] R. Joshi, G. Nelson, and K. Randall, "Denali: A Goal-Directed Superoptimizer," *ACM SIGPLAN Notices*, 2002.

[28] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic Superoptimization," in *SIGARCH Computer Architecture News*, 2013.

[29] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner, "Equality Saturation: A New Approach to Optimization," in *ACM SIGPLAN Notices*, Jan 2009.

[30] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic Binary Translation and Optimization," *IEEE Transactions on Computers*, 2001.

[31] E. R. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the Opportunities of Binary Translation," *Computer*, Mar 2000.

[32] K. Ebcioğlu, E. R. Altman, M. Gschwind, and S. Sathaye, "Optimizations and Oracle Parallelism with Dynamic Translation," in *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999.

[33] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing/spl trade/Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges," in *Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization*, 2003.

[34] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, 1996.

[35] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace Processors," in *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture*, 1997.

[36] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors," in *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, 1998.

[37] Q. Jacobson and J. E. Smith, "Instruction Pre-processing in Trace Processors," in *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999.

[38] S. J. Patel, M. Evers, and Y. N. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[39] S. Patel and S. Lumetta, "rePLay: A Hardware Framework for Dynamic Optimization," *IEEE Transactions on Computers*, 2001.

[40] R. Rosner, M. Moffie, Y. Sazeides, and R. Ronen, "Selecting Long Atomic Traces for High Coverage," in *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.

[41] W. Zhang, S. Checkoway, B. Calder, and D. M. Tullsen, "Dynamic Code Value Specialization Using the Trace Cache Fill Unit," in *Proceedings of the 24th International Conference on Computer Design*, Oct 2006.

[42] S. Patel, D. Friendly, and Y. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism," tech. rep., University of Michigan, 1997.

[43] M. Postiff, G. Tyson, and T. N. Mudge, *Performance Limits of Trace Caches*. 1998.

[44] B. Black, B. Rychlik, and J. P. Shen, "The Block-Based Trace Cache," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

[45] A. Ramirez, J. L. Larriba-Pey, and M. Valero, "Trace Cache Redundancy: Red and Blue Traces," in *Proceedings of the 6th International Symposium on High-Performance Computer Architecture*, 2000.

[46] M. Co, D. A. B. Weikle, and K. Skadron, "Evaluating Trace Cache Energy Efficiency," *ACM Transactions on Architecture and Code Optimization*, Dec 2006.

[47] B. Solomon, A. Mendelson, R. Ronen, D. Orenstien, and Y. Almog, "Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA," in *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, 2001.

[48] D. L. Howard and M. H. Lipasti, "The Effect of Program Optimization on Trace Cache Efficiency," in *Proceedings of the 8th International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[49] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges," in *Proceedings of the 1st IEEE/ACM International Symposium on Code Generation and Optimization*, 2003.

[50] D. Boggs, G. Brown, N. Tuck, and K. S. Venkatraman, "Denver: Nvidia's First 64-bit ARM Processor," *IEEE Micro*, 2015.

[51] I. Kim and M. H. Lipasti, "Implementing Optimizations at Decode Time," in *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[52] M. L. Corliss, E. C. Lewis, and A. Roth, "DISE: A Programmable Macro Engine for Customizing Applications," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[53] M. L. Corliss, E. C. Lewis, and A. Roth, "Low-Overhead Interactive Debugging via Dynamic Instrumentation with DISE," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, Mar 2005.

[54] M. L. Corliss, E. C. Lewis, and A. Roth, "A DISE Implementation of Dynamic Code Decompression," in *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, Jun 2003.

[55] M. L. Corliss, E. C. Lewis, and A. Roth, "Using DISE to Protect Return Addresses from Attack," *SIGARCH Computer Architecture News*, Mar 2005.

[56] M. Taram, A. Venkat, and D. Tullsen, "Mobilizing the Micro-Ops: Exploiting Context Sensitive Decoding for Security and Energy Efficiency," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[57] M. Taram, A. Venkat, and D. M. Tullsen, "Context-sensitive decoding: On-demand microcode customization for security and energy management," *MICRO*, 2019.

[58] M. Taram, D. Tullsen, A. Venkat, H. Sayadi, H. Wang, S. Manoj, and H. Homayoun, "Fast and efficient deployment of security defenses via context sensitive decoding," in *Government Microcircuit Applications and Critical Technology Conference (GOMACTech)*, 2019.

[59] M. Taram, A. Venkat, and D. Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," in *ASPLOS*, Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems.

[60] R. Sharifi and A. Venkat, "CHEx86: Context-Sensitive Enforcement of Memory Safety via Microcode-Enabled Capabilities," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Computer Architecture*, 2020.

[61] J. B. Kotra and J. Kalamatianos, "Improving the Utilization of Micro-operation Caches in x86 Processors," in *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020.

[62] S. Jourdan, R. Ronen, M. Bekerman, B. Shomar, and A. Yoaz, "A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification," in *Proceedings of the 31st Annual IEEE/ACM International Symposium on Microarchitecture*, 1998.

[63] V. Petric, T. Sha, and A. Roth, "Reno: A Rename-Based Instruction Optimizer," in *Proceedings of the 32nd International Symposium on Computer Architecture*, 2005.

[64] B. Fahs, T. Rafacz, S. J. Patel, and S. S. Lumetta, "Continuous Optimization," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, Jun 2005.

[65] A. Perais and A. Seznec, "EOLE: Paving the Way for an Effective Implementation of Value Prediction," in *Proceedings of the 41st Annual International Symposium on Computer Architecture*, 2014.

[66] A. Perais, "Leveraging Targeted Value Prediction to Unlock New Hardware Strength Reduction Potential," in *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021.

[67] W. Zhang, B. Calder, and D. Tullsen, "An Event-Driven Multithreaded Dynamic Optimization Framework," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[68] W. Zhang, D. M. Tullsen, and B. Calder, "Accelerating and Adapting Precomputation Threads for Effcient Prefetching," in *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, 2007.

[69] J. Mars and R. Hundt, "Scenario Based Optimization: A Framework for Statically Enabling Online Optimizations," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2009.

[70] H.-W. Tseng and D. M. Tullsen, "Data-Triggered Threads: Eliminating Redundant Computation," in *Proceedings of the 17th International Symposium on High Performance Computer Architecture*, Feb 2011.

[71] M. DeVuyst, D. Tullsen, and S.-W. Kim, "Runtime Parallelization of Legacy Code on A Transactional Memory System," in *HiPEAC 2011*.

[72] M. Herlihy, J. Eliot, and B. Moss, "Transactional Memory: Architectural Support For Lock-free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 1993.

[73] N. Shavit and D. Touitou, "Software Transactional Memory," in *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995.

[74] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian, "Design and Implementation of Transactional Constructs for C/C++," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, 2008.

[75] N. G. Bronson, C. Kozyrakis, and K. Olukotun, "Feedback-Directed Barrier Optimization in a Strongly Isolated STM," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[76] A. Estebanez, D. R. Llanos, and A. Gonzalez-Escribano, "A Survey on Thread-Level Speculation Techniques," *ACM Comput. Surv.*, Jun 2016.

[77] R. Guo, H. An, R. Dou, M. Cong, Y. Wang, and Q. Li, "LogSPoTM: a Scalable Thread Level Speculation Model Based on Transactional Memory," in *2008 13th Asia-Pacific Computer Systems Architecture Conference*, 2008.

[78] R. Odaira and T. Nakaike, "Thread-Level Speculation on Off-the-Shelf Hardware Transactional Memory," in *2014 IEEE International Symposium on Workload Characterization*, 2014.

[79] J. Salamanca, J. N. Amaral, and G. Araujo, "Using Hardware-Transactional-Memory Support to Implement Thread-Level Speculation," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[80] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[81] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Mar 2009.

[82] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, "I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches," in *Proceedings of the 48th Annual International Symposium on Computer Architecture*, 2021.

[83] M. Taram, X. Ren, A. Venkat, and D. Tullsen, "SecSMT: Securing SMT processors against Contention-Based covert channels," in *Proceedings of the 31st USENIX Security Symposium*, 2022.

[84] M. Lipasti, B. Mestan, and E. Gunadi, "Physical Register Inlining," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[85] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, 2006.

[86] K. Koizumi, K. Hiraki, and M. Inaba, "H3VP: History Based Highly Reliable Hybrid Value Predictor," *1st Championship Value Prediction*, 2018.

[87] A. Seznec, "Exploring Value Prediction with the EVES Predictor," *1st Championship Value Prediction*, 2018.

[88] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Computer Architecture News*, Sept 2006.

[89] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[90] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2010.

[91] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2nd Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Mar 2004.