LoopFrog: In-Core Hint-Based Loop Parallelization

Márton Erdős
University of Cambridge
Cambridge, UK
marton.erdos@cl.cam.ac.uk

Utpal Bora
University of Cambridge
Cambridge, UK
utpal.bora@cl.cam.ac.uk

Akshay Bhosale
University of Cambridge
Cambridge, UK
asb227@cl.cam.ac.uk

Bob Lytton
Arm
Cambridge, UK
bob.lytton@arm.com

Ali M. Zaidi
Arm
Cambridge, UK
ali.zaidi@arm.com

Alexandra W. Chadwick
University of Cambridge
Cambridge, UK
alexandra.chadwick@cl.cam.ac.uk

Yuxin Guo
University of Cambridge
Cambridge, UK
yuxin.guo@cl.cam.ac.uk

Giacomo Gabrielli
Arm
Cambridge, UK
giacomo.gabrielli@arm.com

Timothy M. Jones
University of Cambridge
Cambridge, UK
timothy.jones@cl.cam.ac.uk

Abstract

To scale ILP, designers build deeper and wider out-of-order superscalar CPUs. However, this approach incurs quadratic scaling complexity, area, and energy costs with each generation. While small loops may benefit from increased instruction-window sizes and large loops may see speedups via thread-level parallelism across cores, there remains unexploited medium-granularity parallelism.

We propose LoopFrog to tap into this potential by bringing thread-level speculation schemes into the modern era. LoopFrog runs multiple loop iterations from a single thread in parallel within the microarchitecture. The core can spawn future loop iterations as new microarchitectural threadlets based on compiler-inserted hints, which can leapfrog execution beyond the parent thread's instruction window, exposing a new, medium-grained parallelism, orthogonal to traditional ILP and TLP. LoopFrog monitors data dependencies between executing threadlets, forwards data for true dependencies and squashes speculative threadlets on ordering violations.

Using an LLVM-based compiler to insert hints, we achieve a geometric mean loop speedup of 43%, translating to whole-program speedups of 9.2% on SPEC CPU 2006 and 9.5% on SPEC CPU 2017 benchmarks, with only modest area and power overheads.

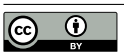
CCS Concepts

• **Computing methodologies** → **Parallel computing methodologies**; • **Software and its engineering** → *Compilers*.

Keywords

parallelism, ILP, speculation, thread-level speculation, speculative multithreading, CPU, multithreading, parallelization, architecture

ACM Reference Format:
Márton Erdős, Utpal Bora, Akshay Bhosale, Bob Lytton, Ali M. Zaidi, Alexandra W. Chadwick, Yuxin Guo, Giacomo Gabrielli, and Timothy M. Jones. 2025. LoopFrog: In-Core Hint-Based Loop Parallelization. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3725843.3756051



This work is licensed under a Creative Commons Attribution 4.0 International License.
MICRO 2025, Seoul, Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1573-0/2025/10
https://doi.org/10.1145/3725843.3756051

1 Introduction

To extract ever larger amounts of ILP, modern CPUs are built with increasingly wide microarchitectures and deep out-of-order instruction windows, incurring significant area and power overheads, as well as growing design complexity. However, single-thread performance scaling exhibits diminishing returns, as complexity/area/power scaling is quadratic with sequential performance [3], further complicating processor design trade-offs in the post-Dennard era [6].

Thread-level speculation (TLS) [27] is a technique originally proposed three decades ago as a means of boosting IPC. A program would be divided into a series of tasks that are logically ordered, but may execute speculatively in parallel. In the event that no data dependencies exist dynamically between tasks, this speculative parallelization would enable a processor to execute programs faster. While this demonstrated promising speedups, fine-grained TLS did not see commercial deployment, with vendors instead preferring wider and deeper cores with more aggressive speculation through long out-of-order windows as a means of boosting ILP. At this point, the proposed benefit of fine-grained TLS has been subsumed in general microarchitectural ILP advancements.

In this paper we revisit TLS in the modern era, using it to expose untapped ILP at *medium* granularity [34], beyond the capabilities of modern out-of-order cores. This different granularity and the highly improved baseline microarchitecture both present new challenges, but also opportunities.

Our approach, LoopFrog, is an in-core microarchitectural loop parallelization scheme, relying on architectural hint instructions. These hints do not change the semantics of sequential execution, but serve to indicate detach and reattach points for light-weight OS-transparent threads, called threadlets, which can speculatively execute work in parallel, thus increasing pipeline utilization. On reaching a detach point, the microarchitecture may choose to copy the current register state to a new threadlet and start its execution

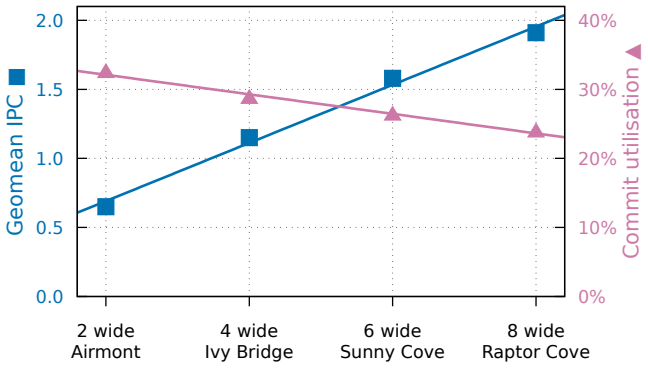


Figure 1: Geomean instructions per cycle (IPC) and commit utilization of SPEC CPU 2017 intrate C/C++ benchmarks measured on four different commercial Intel microarchitectures.

at the reattach point, in parallel with the main execution, using the spare back-end resources of the core. If there are no memory or register conflicts between the threadlet and the main thread, this speculative work can eventually be committed.

Through-memory inter-threadlet data dependencies are handled by a new speculative state buffer (SSB), which sits between the store buffer and the L1 data cache. The SSB forwards memory data from older to younger threadlets and identifies true data dependency violations, which require squashing the younger threadlet and restarting it. Ultimately, threadlets are only retired if no conflicts occur, ensuring LoopFrog preserves the exact same semantics as sequential execution. LoopFrog carefully preserves the architectural memory ordering model, a key requirement for compatibility with modern multicore systems.

We evaluate LoopFrog using the SPEC CPU 2006 and CPU 2017 benchmark suites, showing geometric mean whole-program speedups of 9.2% and 9.5% respectively. Our main contributions are:

- A lightweight ISA extension and execution model, which allow the compiler to expose ordered, speculative task parallelism to the microarchitecture using hint instructions.
- Extensions to a general-purpose CPU microarchitecture to exploit this speculative parallelism at low overhead.
- A detailed characterization of speedups and their causes.

2 Background and Motivation

High-performance microprocessors exploit instruction-level parallelism (ILP) to quickly execute programs. State-of-the-art processors may execute as many as ten instructions per cycle (IPC) on a single thread, and there is a trend towards increasing widths across the industry. Figure 1 shows the geometric mean IPC of the SPEC CPU 2017 intrate C/C++ benchmarks measured on four different Intel microarchitectures. Notably, we see a strong linear relationship between the microarchitecture's designed front-end width and the measured geometric mean IPC. In other words, recent wider microarchitectures are better able to extract ILP from these workloads, increasing instruction throughput.

However, if we consider the percentage of instruction commit bandwidth used for these same processors, we see the opposite trend — the wider architectures have a lower fraction of commit utilization. Given that these wider microarchitectures have more back-end resources, this trend suggests there is an ever increasing amount of unused resources that do not contribute to IPC (most of the time) in newer, wider processors. This presents an opportunity for optimization that we explore in this work: a scheme to use these back-end resources to further boost IPC. If microarchitectures do indeed continue to scale to be even wider in the future, we consider it likely that there will be an increasing amount of under-utilization of back-end resources, so any scheme that can beneficially use these resources will become even more effective.

We build on the idea of thread-level speculation (TLS or SpMT). Originally introduced by the Multiscalar project [27], TLS partitions the program into *tasks* that are speculatively parallelized. If conflicts arise, tasks are *squashed* and subsequently re-executed. We discuss the flavors of TLS in section 7.

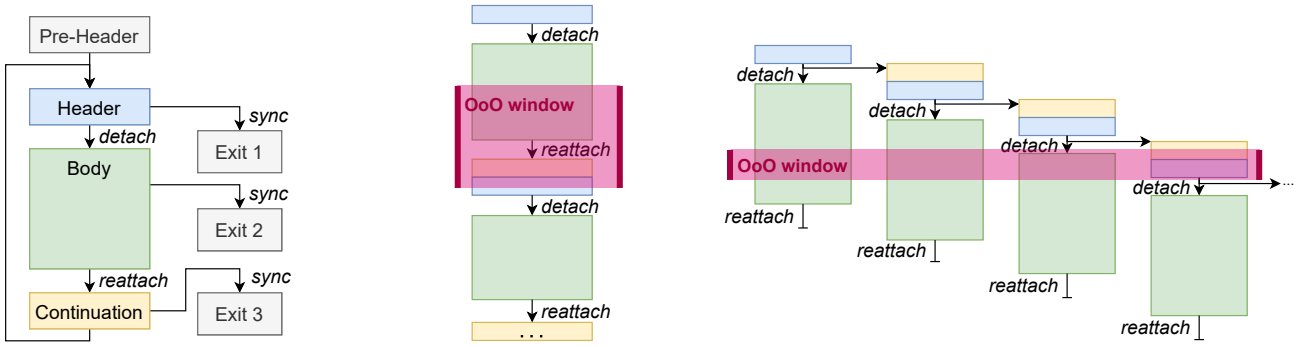
3 LoopFrog Architecture

LoopFrog is an in-core TLS scheme designed to make use of idle back-end resources in wide high-performance microprocessors. LoopFrog does not target the additional parallelism available in a multicore system (across-core parallelization), but instead focuses on maximizing resource utilization within a single core (in-core parallelization). The trend towards wider and deeper microarchitectures to support high ILP peaks provides a performance scaling opportunity for LoopFrog, which can improve core utilization in non-ILP-dominated phases by supplementing them with additional instruction-level parallelism from parallel regions.

We have a strong focus on carefully reducing the barriers to entry for successful deployment. To that end, we keep our approach in-core and hint-based, thus minimizing the impact on the system and retaining already-exploited parallelism. Overall, the programmer-visible semantics of the program are designed to be identical in the case that the LoopFrog hint instructions are treated as `nop`s, allowing architectural backwards compatibility to existing processors.

LoopFrog introduces three hint instructions (`detatch`, `reattach` and `sync`) that are placed in loops. Each iteration of a loop can be seen as a sequence of *header*, *body* and *continuation* sections, the boundaries of which (along with loop exits) are marked with hints, as shown in figure 2(a). The hints do not change the sequential semantics of the program, which must be preserved by the microarchitecture, but instead serve as guidance to enable additional speculation. Sequential execution progresses as shown in figure 2(b), with the out-of-order instruction window (shown as 'OoO window') moving downwards in program order.

The header and continuation contain all register loop-carried dependencies (LCDs), such as induction-variable updates or linked-list traversals. Currently, only loops with relatively simple register LCDs are suitable for parallelization with LoopFrog: no register dataflow is permitted between the body and the continuation, so they both can only consume input register values from their iteration's header. Thus, after executing the header for an iteration, it becomes possible to start executing the body and continuation in parallel, as shown in figure 2(c). This allows the core to execute instructions from a window that is split across multiple quasi-independent regions, increasing the available ILP.



(a) Loop structure and hints (b) Sequential execution (baseline) (c) Parallel execution (LoopFrog)

Figure 2: Static and dynamic view of loops. LoopFrog enables the processor to extract ILP from multiple quasi-independent regions.

The core uses *threadlets* to execute the continuation and subsequent iteration(s) in parallel with the current iteration's body. These are lightweight execution contexts internal to the core, *completely transparent to the operating system and the programmer*. The set of instructions executed by a threadlet is called an *epoch* (one continuation-header-body triple here). Epochs are strictly ordered based on original program order.

Breaking the current iteration's continuation into a separate epoch allows the core to update the simple register LCD values for the next iteration and start executing its header and body in a new threadlet, in parallel with the current iteration's body. The second iteration can then initiate parallel execution of a third epoch to run its continuation in a new threadlet, and so on, until all threadlets in the microarchitecture are utilized.

It is worth noting that the header, body and continuation often do not line up with programming-language or compiler concepts, and instead refer to the sections separated by the *detach* and *reattach* hints. Equivalently, the body is the section that is free from any register LCD updates, and can thus be easily parallelized with future sections. Therefore, theoretically the LoopFrog compiler can insert these hints into every loop, although the size of the resulting body may be small for loops with complex chains of through-register LCDs. An overly small body results in insufficient parallelization and poor performance.

3.1 Parallelization Hint Instructions

For loops that are candidates for parallel execution, the header→body and body→continuation boundaries, as well as loop exits, are annotated using new *branch-like* hints, called `detach`, `reattach` and `sync` (respectively), as shown in figure 2(a). Such hints enable the microarchitecture to identify the boundaries of parallel epochs: an epoch starts at header→continuation and ends at body→continuation.

The proposed hint instructions are inspired from Tapir [23], which embeds asymmetric parallelization directives into LLVM to support *explicit* (i.e. user-specified – Cilk or OpenMP) parallelism in the compiler, without breaking SSA form and most compiler analyses that rely on it (unlike symmetric fork-join directives). However, instead of focusing on explicit parallelism, we utilize similar directives as hints for *implicit* (i.e. speculative) parallelism instead. The machine instructions each carry the continuation block's address, which serves as a unique region ID for each annotated loop.

A `detach` marks a potential fork point: execution can proceed in parallel from here (if the microarchitecture supports it). The current epoch will continue executing from the next instruction, and the successor epoch can be launched in a new threadlet, from the continuation address `C`, executing speculative work. The successor epoch inherits the register state of its predecessor upon `detach`. At this point, the current epoch has 'detached on region `C`', and it will ignore all hints except `reattach C` and `sync C`. If it encounters `reattach C`, then it has caught up to the successor's starting point, and halts. If/once its state has been committed (merged) to architectural state, the successor's state can be committed. Once the current epoch commit succeeds the threadlet can be recycled, making the successor the oldest, non-speculative threadlet. If, on the other hand, `sync C` is encountered instead, then the current threadlet exits the speculative region, which means that the successor threadlet spawning was due to a misspeculation. In this case, the successor is squashed (along with any chained successors), and the current threadlet, once it becomes non-speculative, continues sequential execution of the code after the `sync C`. Thus, the `sync` annotation, on each exit edge, enables early exits from the body by canceling all successors if a given epoch exits the loop. Note that the body is always a contiguous slice of the dynamic instruction stream (even if it is not laid out contiguously in the static binary).

3.2 Preserving Sequential Semantics

Because LoopFrog is an implicit parallelization scheme, the sequential observable semantics of the original serial program must be strictly preserved. The microarchitecture is responsible for maintaining the illusion of sequential execution. Since our epochs are strictly ordered, there is a well-defined total order in which memory operations and side-effects should *appear* to happen logically. Thus, all threadlets must execute *speculatively*, other than the one executing the oldest epoch, which is executing *architecturally*. For speculative threadlets, the microarchitecture must transparently buffer all memory writes (and pause execution before side-effecting

operations like exceptions, barriers or IO) until the threadlet can become architectural, in the correct sequential program order. For speculative threadlets that become architectural, their buffered updates are merged with the global architectural state and are visible to the system in the correct sequential program order. This in-order commit requirement for threadlets eliminates WAW hazards. We implement multi-versioned copy-on-write into speculative buffers for threadlets to eliminate WAR hazards as well. Our proposed scheme is also capable of handling other corner cases of the target architecture memory model, for example, for Arm, RAR hazards to the same memory location.

3.3 Nested Regions

Our architecture permits nested parallel regions (e.g. an inner loop inside the parallel body). Handling such nesting is up to the microarchitecture. Our experiments currently only parallelize one region at a time, using the continuation address as a region ID to identify and ignore hints belonging to inner regions if an outer region is already being executed in parallel. A different implementation may dynamically choose the best nesting level to speculate on, or keep track of profitable/unprofitable region IDs.

Nested parallelization at run time is architecturally permitted with the existing hints. So long as all nested regions have distinct IDs, the microarchitecture can parallelize epochs *fractally* [29], while maintaining total ordering across all epochs. However, given the small number of threadlets in our conventional CPU target, implementing this was not a priority.

4 LoopFrog Microarchitecture

The microarchitecture for executing LoopFrog regions features a high-performance SMT-like pipeline, supporting multiple threadlet contexts (executing parallel epochs). Each threadlet has its own program counter, architectural registers, and logical slice of the ROB, and instructions are tagged with a threadlet ID in the pipeline. Therefore, microarchitectural events such as branch predictions affect instructions only within the same threadlet, and the progress of different threadlets is decoupled at each stage of the pipeline.

As each threadlet executes a different program *epoch*, and epochs are strictly ordered, a total program order is maintained among all instructions and can be established using the epoch number and the instruction sequence number. The oldest threadlet is *architectural*, representing the state of the program, while all other threadlets are *speculative*, and the microarchitecture can freely drop (squash) them at any point if it decides to do so (for example, due to a microarchitectural buffer overflow or a dependency conflict).

In order to preserve the sequential semantics of the program, speculative threadlets are applied (committed) in program order. Furthermore, threadlets *must* be squashed if they *conflict* with a past epoch (that is, if an unhandled dependence violation occurs). In effect, our scheme has two levels of commit. First, instructions commit to their threadlet, and second, speculative threadlets commit to the architectural state of the program once all older threadlets have finished and they are verified to be non-conflicting. Note that each threadlet commits when it becomes the oldest and conflict-checked, and thus it may still have instructions left to execute or commit. Future instructions committed to this architectural threadlet commit directly to the architectural state, as the threadlet cannot be squashed anymore.

In a multicore system, the architecture's memory model imposes restrictions on programmer-observable memory orderings across cores, which LoopFrog explicitly preserves. This requires that speculative threadlets' memory updates must be hidden from other cores until they become architectural. Additionally, threadlets must be squashed if they can no longer be cleanly committed. For example, if another core modifies or observes shared memory in a way that cannot be reconciled with the accesses of the threadlet due to the architecture's memory model.

To track dependency violations between threadlets in a core, all memory accesses update the *conflict detector*, which finds conflicts by tracking the read and write sets of each threadlet. This information is updated in parallel with the access. Memory accesses from the architectural threadlet are dispatched directly to the L1 data cache (L1D) and are externally observable, but they still update the conflict detector (as a speculative threadlet can conflict with them).

Memory accesses from the speculative threadlets are intercepted by the *Speculative State Buffer* (SSB), which sits between the L1D and the memory pipe. The SSB has three main functions. First, it buffers speculatively written data, hiding it from the memory system (and external observers), as well as accesses from older threadlets. Second, it serves up-to-date data to speculative threadlets, acting as a multi-versioned cache. Third, it enables commit and writeback of buffered data without violating memory-ordering constraints. Specifically, the SSB participates in the coherence protocol to facilitate ordered or atomic commit of threadlets, even in the presence of memory traffic from other cores.

Figure 3 shows an overview of the CPU pipeline with resources the threadlets partition, duplicate, or share. Sharing stages is a well-known technique, widely used in simultaneous multithreading processors, thus is not discussed here. Dynamic partitioning can be achieved using linked-list structures [8], as implemented by IBM POWER5 [25] and POWER8 [26]. Older threadlets have priority when making allocations. We add an SSB (see section 4.1), conflict detector (section 4.2), and a checkpoint store to the core. A checkpoint is a snapshot of register state, created when a threadlet starts executing a new epoch. If the threadlet is squashed, we load the checkpoint back in and restart it (if multiple threadlets are squashed, only the oldest one is restarted). Checkpoints can be taken by copying the register rename map and preventing physical registers from being recycled. This can be performed lazily in the background; thus it need not result in a delay.

4.1 Speculative State Buffer

Logically, the SSB holds the values in memory written by each threadlet, addressed by memory address and threadlet ID. Incoming (speculative) memory accesses are tagged with the ID of the threadlet that issued them. Speculative writes store values to the SSB, while speculative reads look up the SSB in parallel with the L1 data cache. The logic in the SSB constructs the most up-to-date data for that threadlet using the result of the SSB and L1D lookup. This multi-versioning logic eliminates output (write-after-write) and

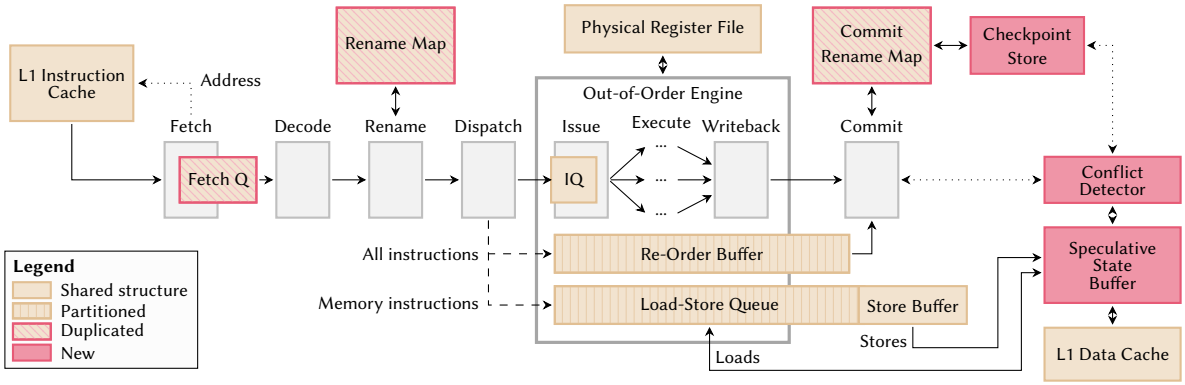


Figure 3: Overview of changes to the pipeline. The LoopFrog microarchitecture shares existing pipeline resources between threadlets, except as indicated. Instructions are tagged with a threadlet ID, and can dynamically co-exist in most structures. The ROB and LSQ are dynamically partitioned into private slices (without expanding them). The fetch queue and rename maps are duplicated to store threadlet-specific information, and three new structures are added for storing speculative data and checking speculation.

false (write-after-read) dependencies, and also eliminates true read-after-write hazards when the read is (correctly) performed after the write, as described in section 4.1.3. Order violations within each threadlet are handled by the load-store unit in the pipeline as usual, true read-after-write violations between threadlets are detected by the conflict detector (see section 4.2), and conflicts between cores are handled by the SSB's coherence logic (section 4.1.4).

The rest of this section discusses the organization of data into *granules* and cache lines (section 4.1.1), the structure of the SSB (section 4.1.2), the versioning logic (section 4.1.3), and the coherence aspects, including threadlet commit (section 4.1.4).

4.1.1 Granules and Cache Lines. Data in the SSB is organized into cache lines composed of small granules. This structure helps to manage metadata, and simplify the conflict checking and access logic.

A *granule* is the smallest unit of data tracked for conflict checking in the SSB. This could be as little as a byte, a word, or up to as much as a full cache line, as defined by the implementation. The cache line size should be a multiple of the granule size. Reads and writes operating on any part of the same granule are considered as overlapping and thus conflicting by the conflict detector. If, for example, the granule size is a word and a store updates a byte in that word, then the SSB executes a read of the whole granule followed by a write with the modification present. The read fills in the unwritten portion with up-to-date data from prior threadlets or main memory. This additional (false) read may lead to a read-after-write conflict, due to false sharing. The larger the granules, the more additional false sharing conflicts will likely occur, but the smaller the conflict checking storage overhead.

A cache line is the unit of allocation and storage within the SSB. Metadata, including address and threadlet ID, is stored per cache line. Each cache line may consist of multiple granules. We store a bitmask per cache line to identify *valid* granules (i.e., granules that the SSB contains) within the line. The SSB only reports a hit if the valid bit is high for a granule targeted by an operation. Upon a speculative store, additional granules may be written to the cache line, and the corresponding valid bits are set. During cache write-back when a threadlet commits, the SSB uses byte-masked writes (or multiple writes if not supported) in case only some granules are valid within the current line.

4.1.2 Structure of the SSB. Figure 4 shows an example of the high-level structure for an SSB supporting 4 threadlet contexts. The SSB features a separate slice for each threadlet, storing values created by that threadlet. For three of the contexts, these values are speculative, and for the fourth one – identified by the S_{arch} counter – they are architecturally visible, thus they respond to coherence requests from the memory system, as described in section 4.1.4.

Squashing is implemented by bulk-invalidating all entries in the affected slices. Speculative writes insert or update entries in the slice corresponding to their threadlet. Speculative reads trigger a parallel lookup for the load address in all SSB slices as well as the first-level data cache (L1D). Together with the S_{arch} counter and the threadlet ID of the load, the results are all fed into the logic described in section 4.1.3 to derive the final, most up-to-date value. Snooped coherence requests trigger a lookup in the architectural slice, and any requested values are flushed to the memory system.

The structure of each slice is also shown in figure 4. The slice is very similar to a standard associative cache. Additionally, each line has a bitmask V identifying valid granules in it, and a hit is only reported if the requested granule's line is present *and* the valid bit for the granule is set. This means the cache can also yield a *partial* hit. A counter is added to track the number of lines present. Once the slice becomes architectural, its data is gradually flushed to the memory system using unused bandwidth. With each flushed line, the counter is decreased. Thus, the slice is fully flushed and the slice can be recycled once the counter hits zero.

A small, fully associative victim cache may be added to each slice to increase effective associativity. Note that speculative writes cannot be discarded or evicted without causing the threadlet to fail. Therefore, the threadlet needs to be stalled or squashed in case the SSB slice is full, or if it cannot take a write due to low associativity. The victim cache reduces the occurrence of the latter case.

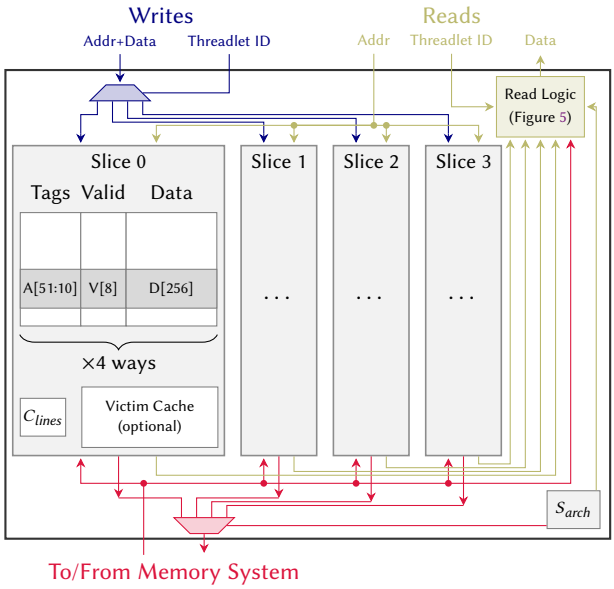


Figure 4: The high-level microarchitecture of the speculative state buffer. The SSB consists of a number of slices storing values for individual threadlets, a read logic circuit, and a counter S_{arch} tracking the slice index of the architectural threadlet. Writes access a single slice only, while reads access all slices, main memory, and S_{arch}. The return value for reads is output by the read logic (see section 4.1.3). Coherence requests from memory are observed by all slices and answered by the architectural slice (see section 4.1.4).

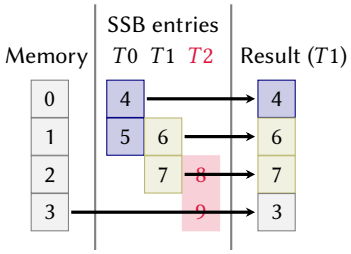


Figure 5: Example lookup of speculative values. The load from threadlet 1 ($T1$) observes the most recent value for each granule, ignoring younger threadlets.

4.1.3 Versioning Logic and Value Forwarding. It is possible to have SSB slices work completely independently, with each operation only performing a lookup in the corresponding slice. However, in this case, we suffer a memory conflict whenever the write set of one threadlet overlaps with the read set of a concurrent threadlet, even if the accesses occur in the correct order. Although these are true read-after-write conflicts, we can eliminate them by performing lookups in all slices, and combining the results carefully. Figure 5 shows the result of this logic. The newest value across the memory system and all older threadlets is returned for each granule, while values from younger threadlets are ignored. This forwarding also ensures that a re-execution of a conflicting read will correctly observe the prior write and thus avoid triggering the same conflict again.

For a speculative load, we can use the threadlet number T of the lookup, and the number of the architectural slice S_{arch} to work out the ordering of threadlets, and thus the values to take. Main memory contains the oldest values, followed by the slice S_{arch}, which corresponds to the architectural threadlet. Then, incrementing the threadlet number (modulo the total number of threadlets) one-by-one yields newer and newer threadlets. Threadlets younger than T are younger than the load instruction. Thus, they contain values created later in program order, and we ignore them during the lookup. This method can be implemented as a simple combinational logic circuit.

4.1.4 Coherence and Commit. By performing speculation, LoopFrog reorders memory operations. Through a combination of multi-versioning, value forwarding, and squashing, we have ensured that the single-threaded semantics of the program are never changed. However, modern architectures also place limits on the reordering of memory operations from the point of view of an external observer. Specifically, LoopFrog must adhere to the memory model, and only reorder memory operations if this is allowed.

To achieve this, LoopFrog never changes the order in which operations are exposed. As the precise ordering between operations from the same speculative threadlet is lost, these are all exposed atomically, at the same time, at threadlet commit. Architectural accesses are exposed immediately, just as during single-threaded execution.

Atomic commit is implemented similarly to hardware transactional memory [11, 13], although deadlocks and livelocks cannot occur in our case (since we only attempt to commit each threadlet once, and continue in architectural mode upon a failure). To facilitate atomic commit, the SSB sends coherence messages during the lifetime of the threadlet to obtain affected cache lines. Lines in the read set are obtained in readable (e.g. *Shared*) state, and lines in the write set in a writable (e.g. *Modified*) state. If another cache later requests a line in an incompatible state, then the line is given up and the threadlet is squashed. To do this, the SSB snoops coherence traffic, and checks requests against the read and write sets of the conflict detector (see section 4.2). Once all lines are obtained and the threadlet becomes the oldest, atomic commit can occur. The S_{arch} counter is incremented in a single cycle, making all lines in the the corresponding SSB slice part of the coherent memory system at once. The SSB slice continues to snoop coherence requests for lines it has (i.e. the part of the write set not yet flushed), and it responds by flushing data immediately so that the requester gets the updated version.

We consider the ability of the SSB to respect the memory model and hide speculation from the memory system crucial for deployability, since doing so maintains the correctness of multi-threaded programs, shared memory mappings and memory-mapped files and devices. Even on relaxed memory architectures (such as Arm or RISC-V), this is necessary, as some limits are still in place on reordering (e.g. between address or data-dependent instructions, or for special memory). Without this commit mechanism, the memory system and architecture would need to be redesigned.

Algorithm 1 Conflict detection logic.

Require: Read and write sets $\text{RD}(T)$, $\text{WR}(T)$ for each threadlet T

```
1:  function SpeculativeRead(Granules G, Threadlet T)
2:      Fwd ← G \ WR(T)    // Derive forwarded set.
3:      RD(T) ← RD(T) ∪ Fwd    // Granule read by threadlet iff forwarded.

4:  function Write(Granules G, Threadlet T)
5:      WR(T) ← WR(T) ∪ G    // Add to write set.
6:
7:      // Check for conflicts
8:      Fwd ← G
9:      for all t from Successor(T) to YoungestThreadlet()
10:         if Fwd ∩ RD(t) ≠ ∅    // t observed stale value!
11:             squash(t)    // Squash and restart t, recycle t + 1, t + 2, ...
12:             break
13:         Fwd ← Fwd \ WR(t)    // All g ∈ WR(t) forwarded from t, not T.
```

4.1.5 Implementation Choices. Note that while we treat the SSB in the preceding discussion and our experimental setup as a distinct microarchitectural component, the functionality it provides could in fact be subsumed by either the store/merge buffers in the core, or by a multi-versioned L1D, if the relative structure sizes allow.

4.2 Conflict Detection

The conflict detector checks the speculation for correctness, and initiates corrective action in the form of squashing threadlets if any speculative reads obtained stale data due to incorrect reordering with prior writes. The conflict detector checks for true read-after-write dependencies between threadlets in the same core, where the read is from a later threadlet but it was serviced *before* the write in the SSB. Note that all other types of conflict are handled without squashing, as described in section 4.1.

To find violations, the conflict detector maintains the read and write sets ($\text{RD}(T)$, $\text{WR}(T)$) for each threadlet T, and performs checks on speculative reads and all writes, as shown in algorithm 1.

Upon a speculative read, no conflict can be discovered (as the read is the latest operation seen), so we simply update the read set of the threadlet. Granules already in the write set have been produced by prior writes in the same threadlet and not writes in prior threadlets. Thus, we exclude these from the threadlet's read set. This is safe, as the values read are always up to date due to ordering within each threadlet.

Upon a write, we add the written granules to the write set for later, and iterate over all younger speculative threadlets t to check for conflicts between the current write W and reads R from future threadlets that were executed before W. The *Fwd* set contains the granules created by W that the read R from threadlet t *should* observe. Before incrementing t, we exclude granules in the write set of t from *Fwd*. To understand why, consider a sequence of accesses to the same location in the program: writes $W_1, \ldots, W_n$ followed by a read R. Clearly, R should (only) observe W_n, as W_n overwrites the values produced by prior writes $W_1, \ldots, W_{n-1}$. If R reads this value, then there is no conflict, otherwise there is a conflict between R and W_n. In the algorithm, consider $W = W_1$. For any granule g, having $g \in \text{WR}(t)$ means there is an intervening W_2, between W and all reads R from threadlets t' younger than t, for granule g. R may or may not conflict with W_n, depending on the timings of accesses, so we do not squash during the current conflict check (from W). Instead, the conflict check initiated by W_n is guaranteed to pick up the conflict if it occurred. If we do identify a conflict, there may still be intervening writes in program order (which have not happened yet), but a conflict is inevitable, and so we squash immediately.

During the lifetime of a threadlet, conflict checking can safely run in the background, without impeding accesses; thus its latency is not critical. However, before the threadlet commits, all checks targeting its reads need to finish. We add a small delay to account for in-progress conflict checks to finish before the threadlet commits.

The read and write sets of threadlets may be implemented in hardware using Bloom filters, similarly to prior work [12]. Doing so leads to a low false-positive rate, but guarantees no false negatives, making the approach safe and efficient.

4.3 Iteration Packing

A number of loops with largely independent iterations are small, containing only tens of instructions on average (sometimes with high variability). If iterations are too small, naïve parallelization will result in a slowdown. For LoopFrog to expose *additional* parallelism, speculative threadlets need to jump ahead of the baseline CPU's out-of-order window, or at least past some hard-to-predict branches. Furthermore, even LoopFrog's thread spawning overheads (e.g. frontend pipeline latency) become significant on such an ultra-low scale.

Thus, to increase the size of each epoch, the microarchitecture can choose to jump further ahead on `detach` and *pack N* loop iterations into each epoch. When detaching from iteration I, the microarchitecture predicts the register starting state of iteration $I+N$. The successor is assigned to execute from $I+N$ onwards, and the current threadlet will execute the iterations until that point.

We use the first few iterations of each loop to train three predictors, which work together to control iteration packing. The first estimates the size of each epoch to derive a reasonable packing factor, targeting a specific epoch size. The second predictor predicts the set of induction variables (IV) (or other loop-carried dependencies) based on cumulative read and write sets across iterations. We treat a register as an induction variable if it is both in the read and write sets (that is, it changes) and the new value is consumed in a later iteration. The third predictor performs value prediction on the suspected induction-variable registers. Iteration packing is only performed if the value predictor can confidently predict all IV registers. The microarchitecture compares predictions to final values, and it safely updates mispredicted registers, or squashes the successor if the stale value has already been consumed.

The general technique of iteration packing can be applied with arbitrarily complex predictors. In particular, we could use any value predictor to handle complicated loop-carried dependencies and find the optimal packing factor. For example, the microarchitecture could execute N copies of the continuation and header (which essentially form the precomputation slice [16]) before entering the body. Note that the microarchitecture still needs to verify either

the final values computed by the p-slice, or its execution, as the p-slice could conflict with the body. We found that even rudimentary predictors suffice to make a difference and tackle the common case.

To set the packing factor, we calculate an exponential moving average (EMA) of past iteration sizes to estimate the likely iteration size S. This can be updated using simple logic $S \leftarrow \alpha S + (1-\alpha)I$, where I is the size of the new iteration, and $0 < \alpha < 1$ is a constant, yet it characterizes common patterns (e.g. constant, phase-based, or noisy sizes) well. We then choose the smallest packing factor P such that $P \times S$ is greater than the ROB size.

To handle the common case of linear iteration, we use a simple strided predictor to predict the value of induction variables. That is, the next value is always predicted as the current value plus a fixed offset. A saturating confidence counter is used to check predictions (with small positive update on success and large penalty on failure). Both the starting value and offset are reset if confidence hits zero. This predictor is simple, quick to train, and covers strided induction variables with occasional conditional updates. We found mispredictions to be rare whenever confidence was high enough for packing to occur. These cases were caught by comparing and verifying register start states. Note that memory conflict detection and disambiguation are also still performed as described in sections 4.1 and 4.2.

5 Compilation and Loop Selection

Although a LoopFrog compiler is not the main focus of this paper, the relatively simple code-generation requirements are a key strength of the technique, especially compared to schemes based on tasks [27, 33], pipelining [19] or auto-parallelization [5, 31].

Code generation for LoopFrog has three main components: loop selection, hint insertion, and transformations to increase parallelism. Our prototype requires manual loop selection using `#pragma` annotations, but performs hint insertion automatically. Optimizing transformations (beyond standard `-O3` passes) are left to future work.

5.1 Loop Selection

In our prototype, we use profiling information to annotate the most profitable loops in the source code, thus simulating perfect static loop selection. The best loops are picked, but the microarchitecture parallelizes all occurrences of them (except those nested in another parallel loop), even if some are unprofitable.

Automatic loop selection is an interesting problem that should be considered both statically and dynamically. Static selection should exclude loops very unlikely to be profitable, while dynamic selection avoids unprofitable parallelization by ignoring hints and treating them as a NOPs. Thus, the overhead of a dynamically de-selected loop is two NOPs per iteration (plus one at the loop exit). This adds costs in very tight inner loops (which should thus be statically deselected), but has negligible impact in larger loops on a wide out-of-order superscalar core. Additionally, unprofitable loops must be excluded by either static or dynamic deselection, as they may lead to slowdown (up to 10% in our tests). Typically, these loops either have very frequent conflicts, overflow the SSB, have low iteration counts, or they already achieve near-maximal performance in the baseline core due to out-of-order execution. A good solution to loop selection will likely use a combination of compiler static analysis, binary profiling techniques, and run-time performance monitoring. The latter may be based on performance counters, or trying both sequential and parallel versions. The boundaries between these components, interfaces for encoding information and possible heuristics present a rich field for future research.

5.2 Optimization

We execute our compiler pass after running all standard compiler optimization passes (`-O3`), including vectorization. In addition to this, we do not perform any LoopFrog-specific optimizations. Such optimizations could increase the size of the parallel body by eliminating or moving loop-carried dependencies. Compiler unrolling could also be used together with code motion to increase the size of iterations and each parallel body.

5.3 Hint Insertion

The compiler pass inserts hints automatically into each selected loop. To do this, it annotates every loop exit edge with a `sync`, and considers possible placements of `detach` and `reattach`. The placement of these hints determines the boundaries of the three loop sections, the *header*, *body* and *continuation*. The *header* is the region above the `detach`, the *body* sits between the two hints, and the *continuation* is the region below the `reattach`. Since the *body* contains the parallel code, we try to maximize its size. However, the compiler must ensure that no true register loop-carried dependence exists between the body and future sections. This excludes some `detach`-`reattach` pairs. The current prototype does not consider through-memory LCDs (except those lowered due to register pressure in the backend) and just blindly maximizes the body size based on profiling information.

6 Evaluation

In this section we show that our implementation is able to take advantage of parallelism exposed by the compiler hints. We demonstrate that this can translate to meaningful whole-program speedups, even for historically hard-to-parallelize general-purpose workloads.

6.1 Setup

We evaluate our proposal for a 4-threadlet configuration using an extended gem5 [2] CPU model, based on an 8-wide OoO microarchitecture described in table 1. All resources are dynamically shared between the threadlets[1], thus our speedups come from better utilization of existing resources. We duplicated the fetch queues per threadlet to better simulate a modern, decoupled front-end. Sharing branch predictor tables appears to work relatively well due to TAGE's ability to learn noisy patterns, although a LoopFrog-aware branch predictor may be beneficial. We do not model the Bloom filters (but account generously for the checking latency), as we believe false aliasing is a second-order effect (due to our fine-grained granules). With a naïve design, this could lead to 2% of epochs failing, but improvements – such as more detailed secondary checks – are possible.

[1] Partitioning statically into 4 threadlets inside each parallel loop (even if only fewer threadlets are alive) produced similar speedups as long as the whole ROB could be used outside of parallel loops.

Core	
Pipeline	4 GHz, 8-wide, dynamically shared by 4 threadlet contexts
Fetch	4 fetch buffers (partitioned 1×4 / 2×2 / 4×1), max 4 insts per buffer, picking $\leq$ 8 insts/cycle/threadlet
Structures	Dynamically shared: 1024-entry ROB, 384-entry IQ, 256-entry LQ and SQ, Duplicated: 32-entry Fetch Queue
Register File	1024 int & 768 fp physical registers
FUs/pipes	7 ALU+Branch, 2 ALU+Mul+Div, 4 SIMD+FP (2 Div/Sqrt/SimdMul + 2 SimdShift/Cvt), 4 Load, 2 Store pipes
Branch Pred	256 Kbits LTAGE [24] (13-component TAGE + 256-entry Loop), 4096-entry BTB, 48-entry RAS. Tables shared and updated by all contexts. (Global) history per threadlet.
SSB	
Work Queue	72-entry (incl. architectural and reattach ops)
Granules & lines	4 B granules, 32 B cache lines
Granule Cache	4 slices, 8 KiB in total, associativity not modelled
Access Latency	1-cycle writes, 3-cycle reads (incl. L1D lookup)
Conflict checking	4-cycle checking latency. No false positives modeled (idealized Bloom filter)
Memory System	
L1I	64 KiB 4-way, 1-cycle hit, 16 MSHRs (×8 targets)
L1D	64 KiB 4-way, 2-cycle hit, 10 MSHRs (×16), 12 write buffers, stride (degree: 2) prefetcher
L2	4 MiB 8-way, 11-cycle hit, 32 MSHRs (×16), 32 write buffers, tagged, stride (degree: 8) and neighbor prefetchers
DRAM	32 GiB DDR3-1600 RAM ($\approx$ 60 ns access latency, 100 GiB/s)

Table 1: Simulation parameters for an aggressive 8-wide core.

The binaries were obtained using the LLVM [14] compiler enabling all `-O3` optimizations, including vectorization, as described in section 5.

We select up to 15 SimPoints [10] per workload, each SimPoint containing 250 million instructions with a warmup period of 50 million instructions to evaluate speedups on full runs of these benchmarks. Clustering yields 8-13 SimPoints for most workloads. We run the gem5 simulation twice per SimPoint. In the baseline run, hints are ignored (no speculation), whereas in the second run, LoopFrog performs speculation, guided by hint instructions. Then, using SimPoint data (representative weights, instructions per SimPoint and in the benchmark), we estimate the run-time with and without LoopFrog speculation. Finally, we divide the total run-times to obtain per-benchmark speedups. We calculate other statistics similarly based on SimPoint weights.

6.2 Whole-Benchmark Speedup

Figure 6 shows our whole-benchmark speedups over SPEC CPU 2006 and CPU 2017 benchmarks. The geometric mean speedup is 9.2% for CPU 2006 and 9.5% for 2017. LoopFrog accelerates 34 out of all 47 benchmarks by more than 1%, including 13 out of 20 in CPU 2017. We focus the rest of our evaluation on these 13 benchmarks. Out of them, 5 achieve over 10% speedup, with *imagick* gaining 87%, *omnetpp* 54%, *nab* 15%, *gcc* 12% and *xalancbmk* 11%. There are also 11 benchmarks from the 2006 suite gaining over 10%.

6.3 Speculation and Region Speedup

Note that the speedups in section 6.2 are over the total run time of the benchmarks, including parallelized loops and sequential regions, which do not get an uplift. To achieve this, our loop speedups are significantly higher, ranging up to 2.9×, with 6 loops achieving over 2× loop speedup, and 44 loops speeding up by 20% or more. Figure 7 shows the degree of LoopFrog threading over the lifetime of each benchmark. We had at least two threadlets active on average 42% of the time in the 13 profitable benchmarks (29% over all benchmarks), and all four threadlets were active 23% of time (16% overall). Using Amdahl's law, this translates to a 43% geometric mean in-region speedup across all CPU 2017 benchmarks.

Figure 8 shows the contributions of architectural execution and speculation to overall performance improvement, as well as additional failed speculation on top. On average, the architectural threadlet is slowed down by 6% compared to the baseline, due to resource sharing. The decrease is limited by LoopFrog always prioritizing the oldest threadlet. Slowdown occurs due to priority inversion on long-term resource allocations (e.g. ROB slots, memory bandwidth) as well as perturbations to predictor inputs. In some cases, we see an uplift in architectural performance, which is due to the prefetching side-effects of speculation that can reduce cache misses, as well as speculative threadlets starting work (typically long-latency cache misses and instructions after them) that only commit after the threadlet becomes architectural. On average, successful speculation recoups the 6% loss and adds the 9.5% speedup. We also see an additional 31% of instructions committed per cycle to speculative threadlets that later fail and do not contribute to architectural IPC. More than two thirds of the failed speculation is contributed by only 5 benchmarks, where these extra instructions could be injected into the pipeline without disrupting architectural progress due to the high degree of under-utilization.

6.4 Analysis

To understand the source of our speedups, we performed additional analysis of bottlenecks and improvements using a wide range of detailed simulator statistics, including top-down microarchitectural analysis [32] metrics. We identified two main classes of speedups, with five subcategories. We then sorted the profitable loops into these categories. While some loops benefit from multiple sources, we focus on the main cause in each case. Because of noise (due to perturbation and secondary effects), we cannot accurately identify the reasons for small changes. Thus, when considering speedup, we limit our analysis to the 38 parallel loops that yielded at least 1% whole-program speedup for their benchmark. A summary is shown in table 2. To estimate the fraction of speedup per category based on this qualitative analysis, we attribute all speedup achieved by each loop to the best-matching category.[2]

6.4.1 True Parallelism. For 28 out of 38 loops, the speedups came primarily from true thread-level parallelism. That is, some of the pipeline's bandwidth was left unused by the single threadlet in sequential mode, or wasted due to misspeculated instructions following branches. In the parallel version, this bandwidth could be reallocated to speculative threadlets, which could use some fraction of it to perform work. As this speculative work was later committed to architectural state, execution speed increased. The natural

[2]For example, the main loop in *omnetpp* benefits mostly from branch condition prefetching, thus all of its speedup is added to this category (even though it benefits to a smaller extent from data value prefetching), since we cannot accurately tell the proportions of those gains.

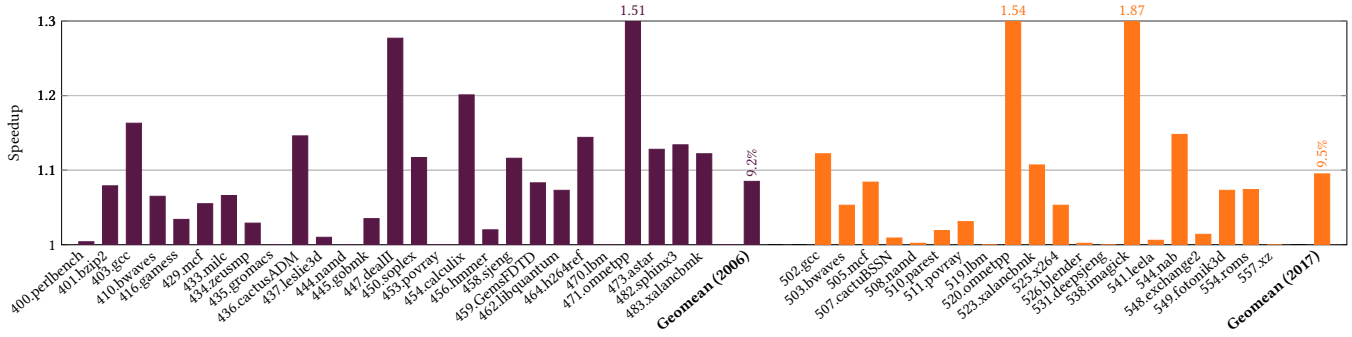


Figure 6: Full-program speedups across SPEC CPU 2006 and CPU 2017 benchmarks (full runs using reference inputs).

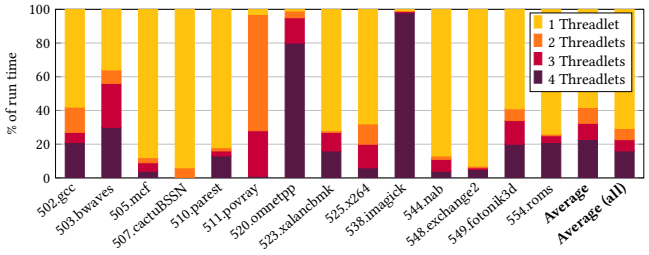


Figure 7: Utilization of speculative threadlets over time for profitable SPEC CPU 2017 benchmarks.

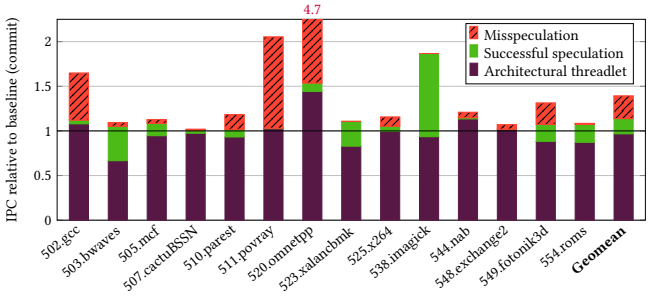


Figure 8: Instructions committed per cycle by the architectural and speculative threadlets (including misspeculation), normalized to baseline IPC.

question is why multiple threadlets could find work better, given that the total number of instructions in the out-of-order windows (e.g. reorder buffer) across all threadlets remain the same as it was in the sequential baseline. We find three main reasons for this.

Memory Parallelism. In highly memory-bound benchmarks, execution speed is primarily determined by how quickly the processor can discover and execute long-latency memory instructions. Single-threaded execution is characterized by alternating bursts of higher-IPC, more compute-bound work, and long low-IPC phases, waiting for stalled memory instructions before their dependent instructions can be executed. During these low-IPC phases, most or all independent instructions in the window from the stalled accesses have already been executed, and the window typically cannot extend further due to a stalled load at the front. Thus, the pipeline sits highly underutilized and performing work from speculative threadlets is very cheap. During low-IPC phases of the oldest threadlet, speculative threadlets can run ahead to discover more high-latency memory instructions, far in the future, and initiate the accesses. This early discovery leads to timely dispatch, as well as an increase in memory-level parallelism. This is because far away accesses in the instruction stream are more likely to be independent due to the way code is structured, thus the instruction windows (on average) contain more accesses that are ready to go. Note that this is despite the total number of instructions in the windows remaining the same. Additionally, having up to 4 streams of instructions with opposing phases naturally lining up creates a more steady instruction mix, reducing contention on core resources such as integer or floating-point ALUs, or vector execution pipes. Memory access latency dominates performance in 17 profitable loops, accounting for 29% of our gains. Additionally, memory-level parallelism plays an important role in many other loops.

Cutting Control Dependencies. Since threadlets have independent ROB slices, branch mispredictions in one threadlet do not affect the other threadlets. Thus, the hints encode high-level control flow predictions, breaking some (true) control flow dependencies. On top of predicting a future re-convergence point in the program, the implicit predictions encoded by LoopFrog's hints (same value of registers, and independence on memory) allows early execution and speculative commit of instructions in the continuation. We found this to be the main bottleneck in 9 loops, which together yield 23% of the total speedup from our scheme.

Cutting Dependency Chains. Similarly, the pipeline can also run out of independent work within the window due to long chains of linearly dependent instructions, even if they do not have high latency individually. Since there is no instruction-level parallelism available between instructions that (directly or indirectly) depend on each other, a single instruction window can only expose limited ILP in programs dominated by long dependency chains. As instructions further away in the stream are statistically less likely to be dependent, we expose additional ILP due to looking at multiple far-away subwindows. This is the main gain in 2 loops, accounting for 12%, however, we also observe the same effect across all three 'true parallelism' categories, likely contributing to more speedup.

Category	Sub-category	Loops	Fraction of Speedup
True parallelism	Memory parallelism	17	29%
	Control dependencies	9	23%
	Dependency chains	2	12%
Prefetching	Branch conditions	6	32%
	Data values	4	3%

Table 2: Sources of performance gains. For each source, we show the number of loops where it is the main factor, and the fraction of the total geometric mean speedup achieved by those loops across all SPEC CPU 2017 benchmarks.

6.4.2 Prefetching. Ten loops (in *omnetpp*, *gcc*, *povray* and *exchange2*) had minimal or no successful speculation, but still achieved a performance uplift due to the side effects of failed speculation. Specifically, since speculative accesses bring data into the L1 cache, LoopFrog can serve as a sophisticated prefetcher. Although sometimes this prefetching yields benefits by speeding up the delivery of data to general computations (3% of speedup), we found the main benefit (32% of total) comes from faster computation of branch conditions. This in turn led to faster execution of hard-to-predict, data-dependent branches, which in turn helped earlier redirection to the correct path, leading to fewer misspeculated instructions, lower front-end pressure, and higher average ROB occupancy. Consequently, higher levels of instruction-level parallelism were present in the window.

This distribution is perhaps not surprising, as it is easy to mask cache-missing loads using ILP if independent work exists within the window, but the same would typically not be possible for mispredicting branches, since all future instructions are control-dependent on the branch. Since modern branch predictors are very good at predicting most branches, tackling the remaining bottleneck of data-dependent branches can have a significant impact on performance.

6.4.3 Benchmarks With No Speedup. Benchmarks *namd*, *lbm*, *blender*, *deepsjeng*, *leela*, and *xz*, showed little or no speedup with the current scheme. High-coverage loops in these benchmarks tend to fall into one of several categories: they are either extremely large (*lbm*, *xz*) or very small (*leela*) in size, have a low trip count (*blender*, *deepsjeng*), demonstrate high IPC with a saturated pipeline (*deepsjeng*, *namd*), or involve frequent cross-iteration dependencies that require more advanced DoAcross synchronization.

6.5 Iteration Packing

The iteration-packing optimization affects 5 of the 13 profitable benchmarks, increasing the overall geometric mean speedup by 0.9 percentage points across SPEC CPU 2017. The mean packing factor is 2.1×, with a maximum of 25×. Since this is a fairly heavy-weight optimization (requiring value prediction and extra register tracking), a microarchitecture may choose to omit it, (and still gain 8.6%). Compiler unrolling and code motion may also be a more light-weight alternative.

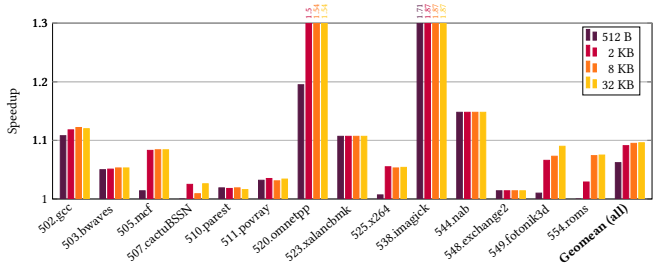


Figure 9: Sensitivity to SSB size (default: 8 KiB).

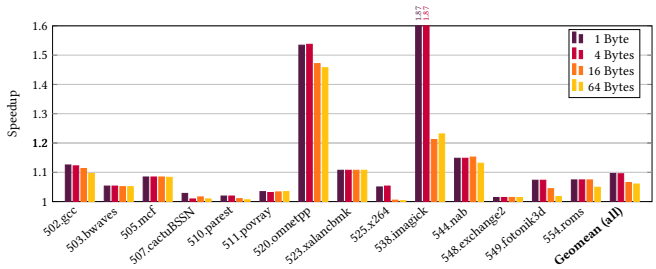


Figure 10: Sensitivity of speedups to SSB and conflict detector granule size (default: 4 bytes).

6.6 Sensitivity to SSB Parameters

We investigate how changing two crucial parameters of the SSB and conflict detector affect performance.

Figure 9 shows sensitivity to SSB size. We find that speedups are surprisingly resistant to change even with large changes in size. Changing the size from the headline 8 KiB to 32 KiB increases the geometric mean speedup by less than 0.1%, and reducing to 2 KiB only has a 0.4% impact. We find that size affects loop speedups in a fairly binary way: if the working set fits in the SSB, we can successfully parallelize, otherwise we lose most parallelization gains for the loop. Whilst decreasing the size to 512 B does hurt speedups far more, we find it noteworthy that this setup still gains 6.2%. Notably, with our 4-byte granules, we only store up to 128 entries in the SSB, which is only half of the simulated load-store queue's capacity.

Figure 10 explores the effects of granule size, which affects both the storage in the SSB, as well as checking logic in the conflict detector. We find that scaling up to half-word (4 B) granularity has no measurable impact on performance beyond noise for any of the benchmarks. Moving to word size (8 B) only induces a slowdown for one benchmark (5% for *x264*), however increasing further introduces far more false aliasing, resulting in slowdowns. For 16 B granules, speedups drop to 6.5%, and then to 6% for cache-line-granularity conflict checking. Note that this is the approach typically used by related work [12, 20, 28], so our granular conflict checking is a key novelty.

Low cache associativity can affect the results negatively. We ran CPU 2017 benchmarks with SSB associativity limited to 4- and 8-ways across all threadlets. This resulted in a performance hit of 2.0% and 1.4%, respectively, compared to our headline results. However, adding a small victim buffer (8 entries, shared) reduced the impact

Scheme	LoopFrog	STAMPede (private cache) (2005)	MultiScalar (1995)
Speedup	1.1× (SPEC 2017)	1.16× (subsets of SPEC 1995 and 2000)	2.16× (SPEC 1992)
Cores	1 (4-way SMT)	4	8 (PUs)
Area	~ 1.15×	> 4×	~ 8×
Baseline	8-issue OoO	4-issue simple OoO, 5 stages	2-issue limited OoO (ROB=32)
Task sizes	~100–10,000 instructions	~1,400 instructions	10–50 instructions
Deployment	compiler, ISA hints	OS, compiler, ISA	specialist μ-arch, compiler, ISA

Table 3: Comparison of LoopFrog with comparable TLS/SpMT schemes [27, 28]. Speedup numbers are not like-for-like due to wildly different baseline cores, different benchmark sets, and area overheads.

to 1.2% in both cases. This slowdown is almost exclusively suffered by *omnetpp* (6.9% slower) and *imagick* (8.8% slower).

6.7 Generality of LoopFrog

LoopFrog works well on loops with different characteristics. Among the profitable loops, some contain inner loops, irregular control flow, function calls, and some are vectorized. By targeting irregular, medium-grained parallelism, LoopFrog remains highly orthogonal to existing ILP, DLP and TLP techniques. None of our loops had OpenMP parallel pragmas, although some had pragmas on outer loops. In these cases, even if parallelization were turned on for the coarse outer loop, LoopFrog can still deliver finer-grained parallelism by parallelizing the inner loop. On the SPEC CPU 2017 benchmarks, considering only loops that are not within an OpenMP parallel region, LoopFrog gains a geometric mean speedup of 7.5%.

6.8 Area and Power Overheads

Our main overheads are due to the addition of threadlets to the pipeline, the SSB, and the conflict-detection logic. Speculation increases the number of instructions issued by 14%, while L2 cache accesses only increase by 1.7%, and L2 misses actually decrease by 2.3%. This may occur due to shallower intra-threadlet speculation and alignment in accesses, and likely yields a small negative change in power usage from speculation.

Handling threadlets is a simplified form of simultaneous multithreading (SMT), which requires a 10–15% increase in area and power [4, 9, 15]. The SSB's primary storage area is the granule cache. The four slices, each with 2 KiB capacity and 4-way cache associativity, correspond to an approximate area of 0.025 mm^2 (0.03 nJ per access) according to CACTI [18] (22 nm, `itrs-hp`, 1 read/write port, 1 read-exclusive port). In a 7 nm node, using a conservative scaling factor of 5, the total area for the four slices is 0.02 mm^2. Finally, an implementation of conflict checking based on Bloom filters, similar to Swarm [12], takes approximately 0.005 mm^2 of area at 7 nm (dual-ported SRAM with 8 entries, and 4,096-bit filters). The area of all additional components is around 2% compared to the total area of a relatively high-end core, for example, an Arm Neoverse N1 (1.4 mm^2 at 7 nm, including private L1 and 1 MB L2 caches [22]). Hence, we expect a total increase of 12–17% in area compared to a sequential design, and only about 2% if SMT support already exists. Even with the optimistic Pollack's rule approximating traditional performance gains from a 12–17% increase at 6–8%, LoopFrog's headline speedup of 9.5% outperforms expected scaling.

7 Related Work

LoopFrog is built on the idea of thread-level speculation, which has been explored by a wealth of past papers [7, 30]. LoopFrog's novelty is to encode TLS using ISA hints without exposing speculation to the OS or memory system, and use it to tackle low resource utilization in a wide out-of-order superscalar processor. This enables the technique to be incorporated into modern high-performance systems. In contrast, past TLS work focused on multicore or core federation (thread/processing unit) approaches which involves OS and architectural impact, or old SMT cores.

Early approaches [16, 17, 27] use a ring of thread units (or processing units) to map computations, with shared speculative memory (and register) state. Compared to today's cores the thread units are simple and have low performance. Thus speedups compared to the low-powered baseline must compete with 20+ years of microarchitectural improvements, including the slowdowns suffered on non-parallel regions, where sequential performance matters.

Another option is to scale the approach to a multicore design. Unfortunately, this increases the communication latency. Several approaches [20, 28] target more coarse-grained parallelism to amortize launching and synchronization overheads. Due to the different granularity, we see this as largely orthogonal to LoopFrog; indeed, the two schemes could both run on the same system. The alternative (explored by Swarm [12, 33]) is to double down on small tasks. Specialized logic is required to speed up communication, and (in both cases) there is significant impact on the ISA and operating system. Furthermore, speculation is exposed to the memory system, rendering TLS incompatible with traditional multi-core execution. Due to the number of cores used and special hardware requirements, these schemes have a very high area cost compared to a single core. Furthermore, deployment would be extremely tricky due to incompatibility with traditional multicore architectures and memory systems. Code generation is also tricky, leading to only half of the SPEC CPU 2006 benchmarks compiling due to exception and control-flow limitations of the prototype.

Using TLS in a simultaneously multithreaded (SMT) core – like LoopFrog does – has been proposed before [1, 21]. However, past schemes focus on parallelism found within tens to hundreds of consecutive program instructions, relying on the LSQ (and ROB) for reordering and memory disambiguation. We believe the gains achieved have been superseded by progress in CPU core design.[3]

[3] IMT [21] only achieves IPC 1.4 on SPEC CPU 2000 on an 8-wide core, which is worse than our measurements show on a 4-wide Intel© Xeon© W-2195 machine. IMT also claims that DMP [1] produces a slowdown.

We compare LoopFrog to the most relevant past TLS schemes listed in table 3. It is hard to compare numerically to past schemes due to highly varied baseline hardware, unavailable artifacts, and old/limited benchmarks. We found most past work to only use a limited set of workloads either due to compiler limitations (suggesting a complicated compiler deployment story), poor performance, or both. Furthermore, many schemes have a far higher hardware footprint, requiring a specialized re-design of the whole chip or the memory system. We believe LoopFrog's strengths lie in its ability to keep the compiler simple, have no OS impact, and limit hardware changes and speculative state to the inside of a single core.

Increasing core utilization to gain performance has been explored by pipelining and helper threading techniques [31]. Recently, Pipette [19] added architectural support for structuring code into pipelines with stages decoupled by queues. The authors demonstrated 1.9× speedup in select irregular applications using 4-way SMT. However, they acknowledged that the transformations are challenging to automate, therefore Pipette requires changes to the programming model. Furthermore, Pipette requires complete independence between stages, unlike LoopFrog, restricting the set of target applications.

Recent work by Eyerman et al. [8] used Tapir-like hints to identify quasi-independent regions in order to enable selective flushing on branch mispredicts, and achieve 1.3× (harmonic mean) speedup over graph applications. However, the speedups depend on high mispredict rates in addition to task-level parallelism, reducing the applicability of the technique to only a smaller subset of workloads LoopFrog can parallelize.

8 Conclusion and Future Work

We identified untapped parallelism, missed by ILP and TLP schemes in state-of-the-art high-performance cores. We proposed a minimal, hint-based ISA extension that can mark speculatively parallel regions, and placed them using a prototype compiler. We designed a low-overhead, in-core microarchitectural extension to take advantage of these hints. Our scheme can be cleanly added to existing high-performance processors without breaking compatibility. We demonstrated that our technique can effectively improve performance on hard-to-parallelize general purpose workloads, achieving a full-program geometric mean speedup of 9.2% (SPEC CPU 2006) and 9.5% (SPEC CPU 2017), speeding up parallel loops by 43%. Future work will focus on microarchitectural techniques and compiler transformations to better handle loops with complex cross-iteration dependencies, taking advantage of the substrate introduced in this paper.

Acknowledgments

This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), grant EP/W00576X/1, and Arm. Additional data related to this publication is available in the repository at https://doi.org/10.17863/CAM.120739.

References

[1] Haitham Akkary and Michael A. Driscoll. 1998. A Dynamic Multithreading Processor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, James O. Bondi and Jim Smith (Eds.). ACM/IEEE Computer Society, 226–236. https://doi.org/10.1109/MICRO.1998.742784

[2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011). https://doi.org/10.1145/2024716.2024718

[3] Shekhar Borkar. 2007. Thousand Core Chips: A Technology Perspective. In *Proceedings of the Design Automation Conference (DAC)*. https://doi.org/10.1145/1278480.1278667

[4] J. Burns and J.-L. Gaudiot. 2002. SMT Layout Overhead and Scalability. *IEEE Transactions on Parallel and Distributed Systems* 13, 2 (2002). https://doi.org/10.1109/71.983942

[5] Simone Campanoni, Timothy M. Jones, Glenn Holloway, Gu-Yeon Wei, and David Brooks. 2012. HELIX: Making the Extraction of Thread-Level Parallelism Mainstream. *IEEE Micro* 32, 4 (2012). https://doi.org/10.1109/MM.2012.50

[6] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/2000064.2000108

[7] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. A Survey on Thread-Level Speculation Techniques. *Comput. Surveys* 49, 2 (2016). https://doi.org/10.1145/2938369

[8] 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 International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3466752.3480045

[9] Josue Feliu, Stijn Eyerman, Julio Sahuquillo, and Salvador Petit. 2016. Symbiotic Job Scheduling on the IBM POWER8. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2016.7446103

[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). http://www.jilp.org/vol7/v7paper14.pdf

[11] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, New York, NY, USA, 12 pages. https://doi.org/10.1145/165123.165164

[12] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A Scalable Architecture for Ordered Parallelism. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2830772.2830777

[13] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA.2018.00045

[14] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2004.1281665

[15] Yingmin Li, K. Skadron, D. Brooks, and Zhigang Hu. 2005. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*. https://doi.org/10.1109/HPCA.2005.25

[16] Carlos Madriles, Carlos García Quiñones, F. Jesús Sánchez, Pedro Marcuello, Antonio González, Dean M. Tullsen, Hong Wang, and John Paul Shen. 2008. Mitosis: A Speculative Multithreaded Processor Based on Precomputation Slices. *IEEE Transactions on Parallel Distributed Systems* 19, 7 (2008), 914–925. https://doi.org/10.1109/TPDS.2007.70797

[17] Pedro Marcuello, Jordi Tubella, and Antonio González. 1999. Value Prediction for Speculative Multithreaded Architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Ronny Ronen, Matthew K. Farrens, and Ilan Y. Spillinger (Eds.). ACM/IEEE Computer Society, 230–236. https://doi.org/10.1109/MICRO.1999.809461

[18] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009). https://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf

[19] Quan M. Nguyen and Daniel Sanchez. 2020. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO50266.2020.00056

[20] M. Ohmacht, A. Wang, T. Gooding, B. Nathanson, I. Nair, G. Janssen, M. Schaal, and B. Steinmacher-Burow. 2013. IBM Blue Gene/Q memory subsystem with speculative execution and transactional memory. *IBM Journal of Research and Development* 57, 1 (2013). https://doi.org/10.1147/JRD.2012.2228092

[21] Il Park, Babak Falsafi, and T. N. Vijaykumar. 2003. Implicitly-multithreaded processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Allan Gottlieb and Kai Li (Eds.). https://doi.org/10.1109/ISCA.2003.1206987

[22] Andrea Pellegrini, Nigel Stephens, Magnus Bruce, Yasuo Ishii, Joseph Pusdesris, Abhishek Raja, Chris Abernathy, Jinson Koppanalil, Tushar Ringe, Ashok Tummala, Jamshed Jalal, Mark Werkheiser, and Anitha Kona. 2020. The Arm Neoverse N1 Platform: Building Blocks for the Next-Gen Cloud-to-Edge Infrastructure SoC. *IEEE Micro* 40, 2 (2020). https://doi.org/10.1109/MM.2020.2972222

[23] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2017. Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*. https://doi.org/10.1145/3018743.3018758

[24] André Seznec. 2007. A 256 Kbits L-TAGE branch predictor. https://www.irisa.fr/caps/people/seznec/L-TAGE.pdf.

[25] Balaram Sinharoy, Ronald N. Kalla, Joel M. Tendler, Richard J. Eickemeyer, and Jody B. Joyner. 2005. POWER5 system microarchitecture. *IBM Journal of Research and Development* 49, 4-5 (2005), 505–522. https://doi.org/10.1147/RD.494.0505

[26] Balaram Sinharoy, James Van Norstrand, Richard J. Eickemeyer, Hung Q. Le, Jens Leenstra, Dung Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, José E. Moreira, D. Levitan, S. Tung, David Hrusecky, James W. Bishop, Michael Gschwind, Maarten Boersma, Michael Kroener, Markus Kaltenbach, Tejas Karkhanis, and K. M. Fernsler. 2015. IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development* 59, 1 (2015). https://doi.org/10.1147/JRD.2014.2376112

[27] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. 1995. Multiscalar Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/223982.224451

[28] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. 2005. The STAMPede Approach to Thread-Level Speculation. *ACM Transactions on Computer Systems* 23, 3 (2005). https://doi.org/10.1145/1082469.1082471

[29] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. 2017. Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/3079856.3080218

[30] Josep Torrellas. 2011. *Speculation, Thread-Level*. Springer US, Boston, MA. 1894–1900 pages. https://doi.org/10.1007/978-0-387-09766-4_170

[31] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. 2007. Speculative Decoupled Software Pipelining. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE Computer Society, 49–59. https://doi.org/10.1109/PACT.2007.66

[32] Ahmad Yasin. 2014. A Top-Down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 35–44. https://doi.org/10.1109/ISPASS.2014.6844459

[33] Victor A. Ying, Mark C. Jeffrey, and Daniel Sanchez. 2020. T4: Compiling Sequential Code for Effective Speculative Parallelization in Hardware. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1109/ISCA45697.2020.00024

[34] Ali Mustafa Zaidi, Konstantinos Iordanou, Mikel Luján, and Giacomo Gabrielli. 2021. Loopapalooza: Investigating Limits of Loop-Level Parallelism with a Compiler-Driven Approach. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*. https://doi.org/10.1109/ISPASS51385.2021.00030