In-core, hint-based, speculative multithreading

UNIVERSITY OF CAMBRIDGE

Marton Erdos Alexandra Chadwick Utpal Bora Akshay Bhosale Yuxin Guo
Timothy Jones Giacomo Gabrielli (ARM) Ali Zaidi (ARM)

PARASOL

I. Motivation: single-threaded scaling limits

Out-of-order window size: rapidly increasing

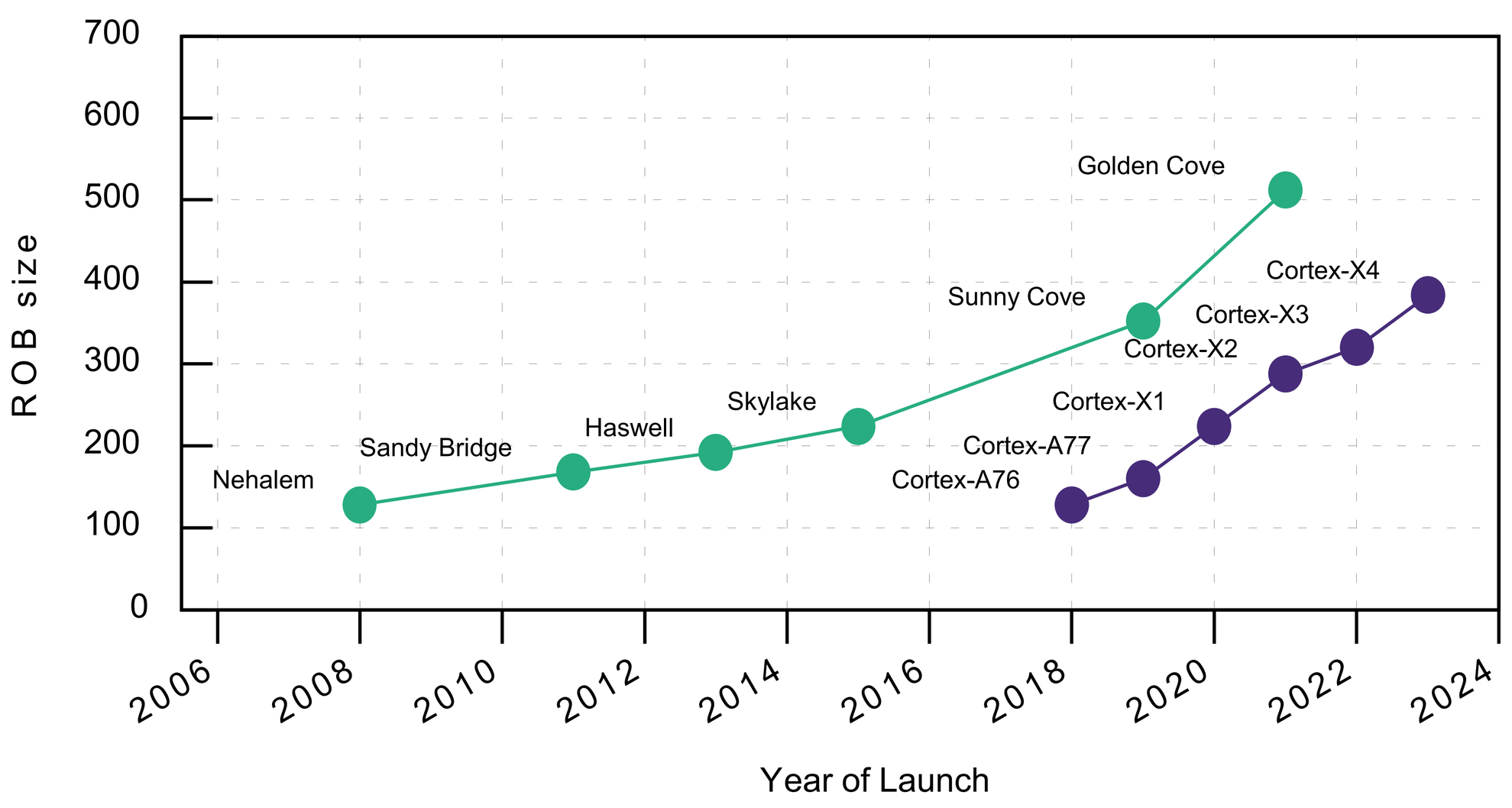
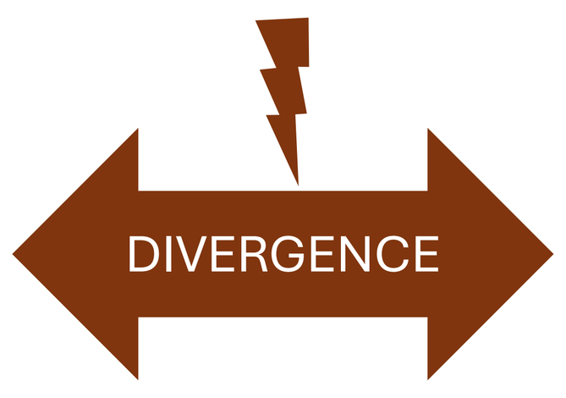


Figure 1.1. Out-of-order windows are rapidly getting longer in order to find more instruction-level parallelism (ILP).



Single-threaded performance: tailing off

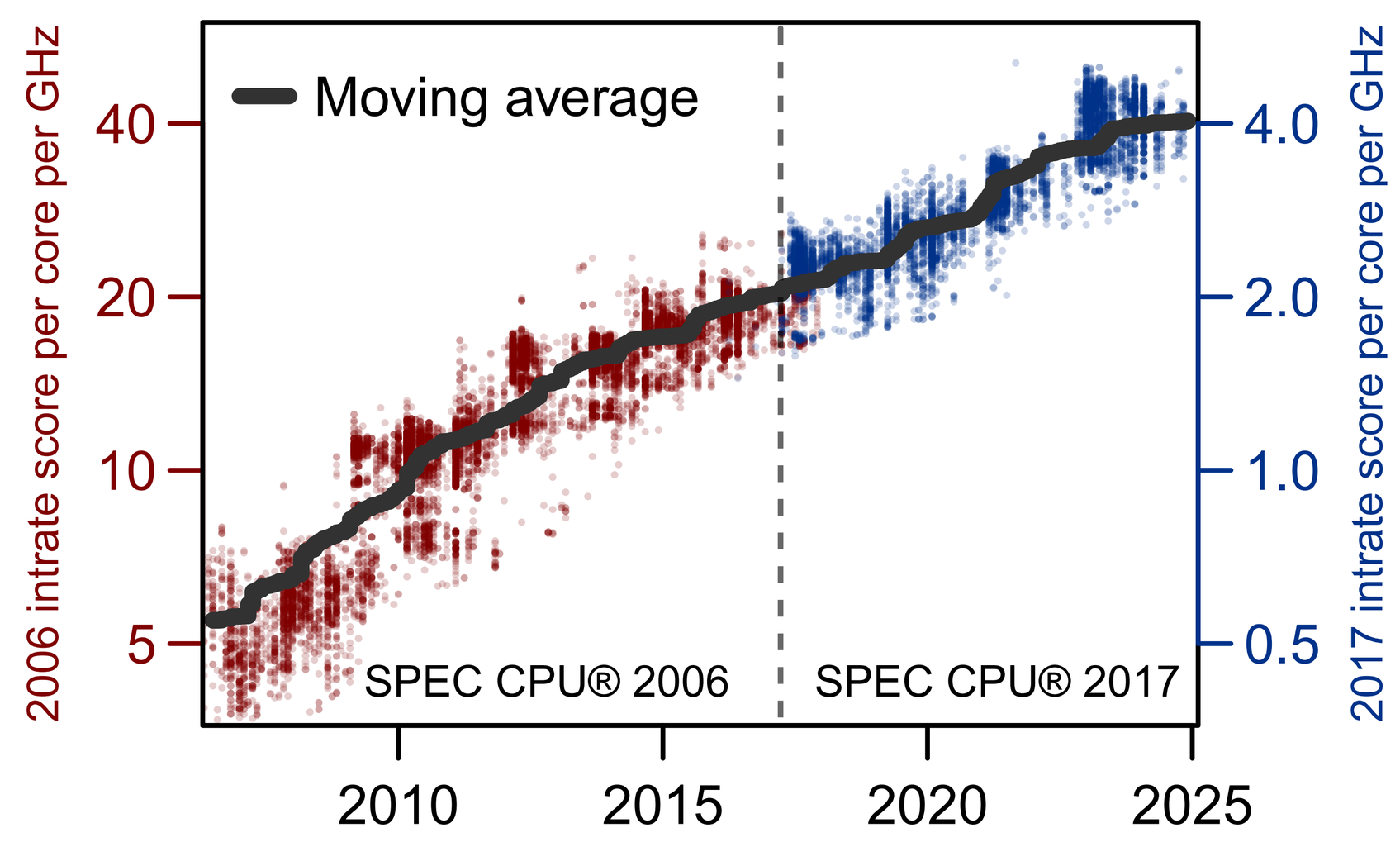


Figure 1.2. SPEC® CPU 2006 and 2017 integer rate scores scaled by frequency and core count (all submissions). Used as a proxy for single-thread performance.
Source: spec.org

CPU pipelines: underutilised

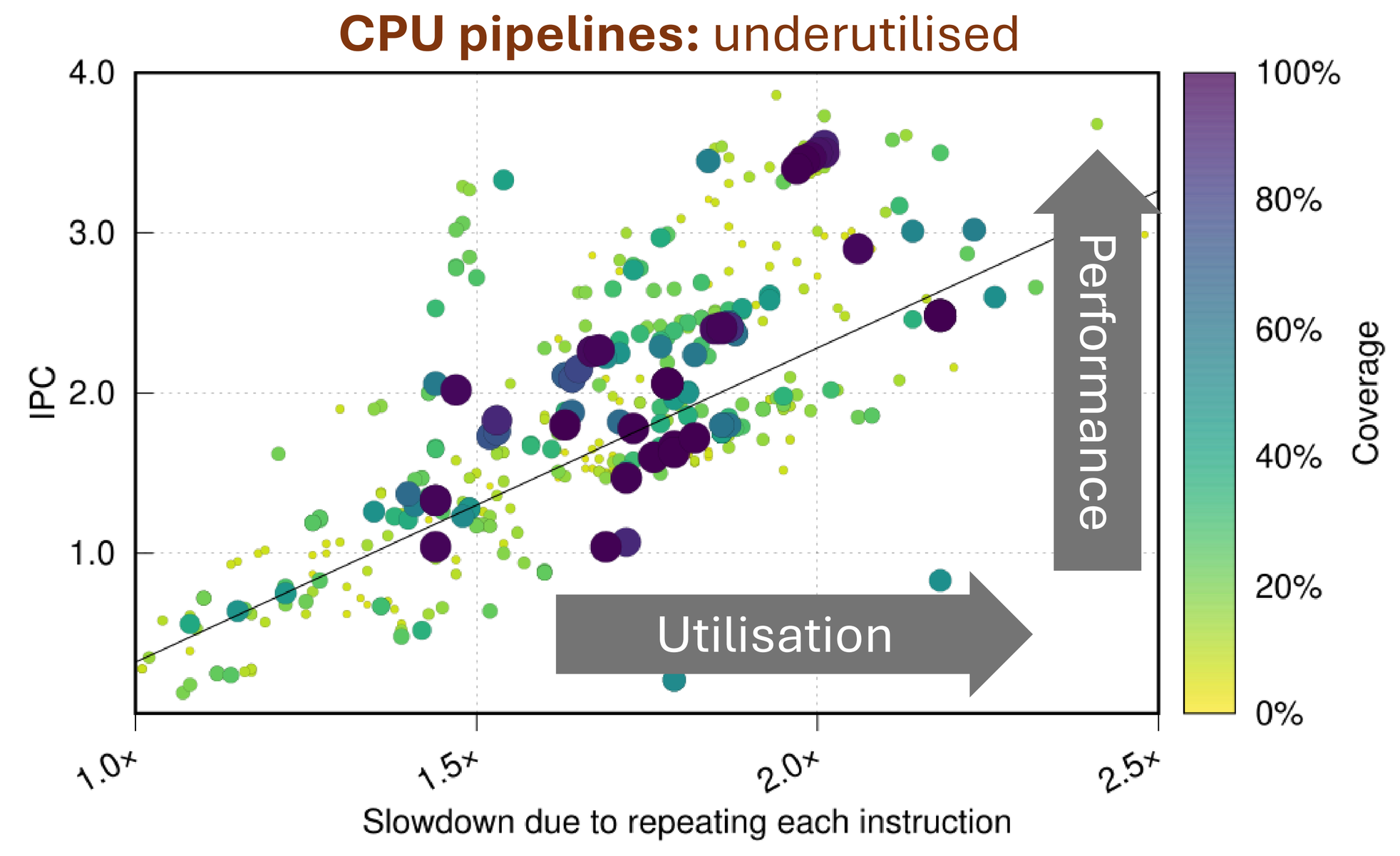


Figure 1.3. Slowdown of SPEC CPU2006 loops from duplicating each instruction (approximating utilisation) against base IPC. Run on a 4-wide Intel® Xeon® W-2195 CPU from 2017.

Modern out-of-order superscalars are *wider* and speculate *deeper* into the program than ever (Fig 1.1) to find instruction-level parallelism. This has lead to diminishing returns in performance (Fig 1.2), due to under-utilisation of pipeline resources (Fig 1.3).

We proposes to apply the idea of thread-level speculation (TLS) to a single modern CPU core, based on lightweight hint instructions, in order to *expose more parallelism* using speculative jumps. Our co-design approach aims to ensure compatibility and ease potential adoption in practice.

II. Architecture: hints

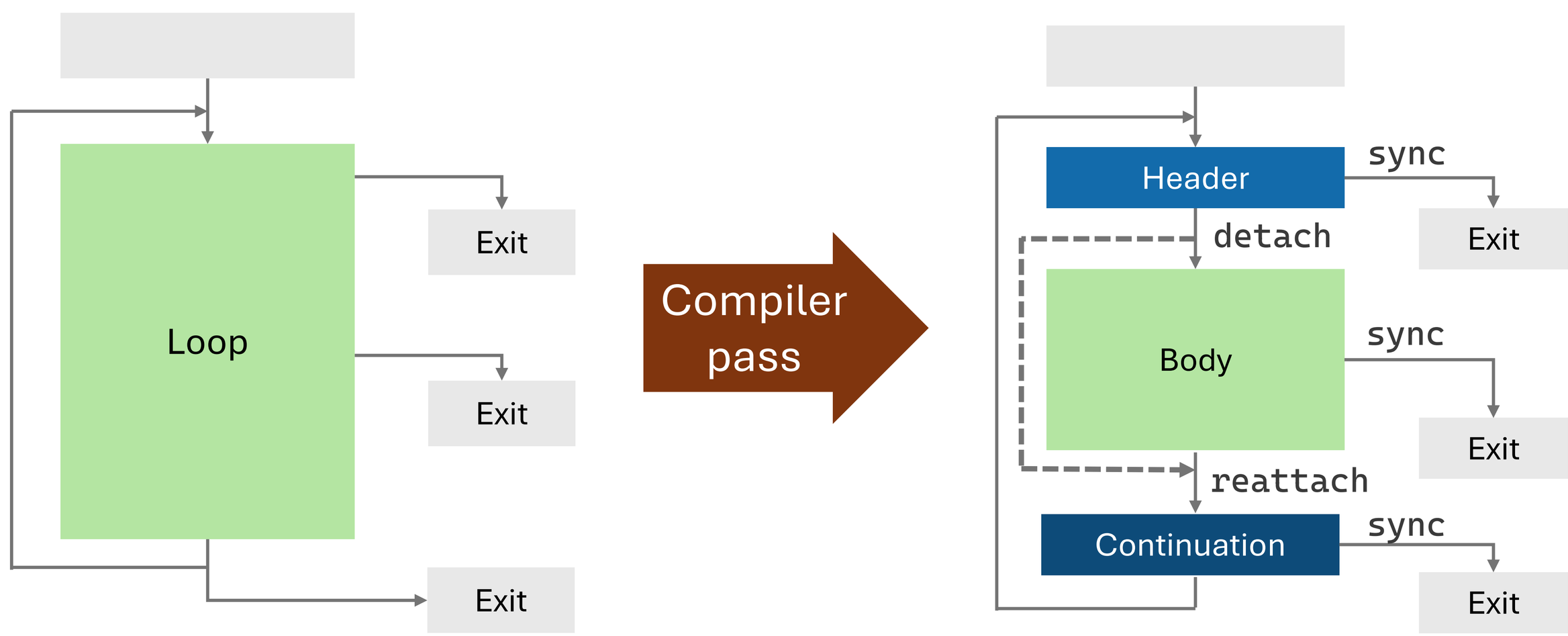


Figure 2.1: Hints (î ì œğř ǒǰ ėğėğř ǑǖŬğ) are added by the compiler.
Detach and *reattach* mark the *potentially-parallel* body, and *sync* marks the exits.

Three *hint instructions*, called *detach*, *reattach*, and *sync* mark the bounds of speculation (Fig 2.1). These expose a possible parallel schedule (Fig 2.2).

The hints do not change sequential semantics. They can be safely ignored, leading to sequential execution.

Parallel semantics: The compiler *says* that

- all *live registers* likely have the *same value* between *detach* and *reattach*, and
- the body *may be memory-parallel* with future sections.

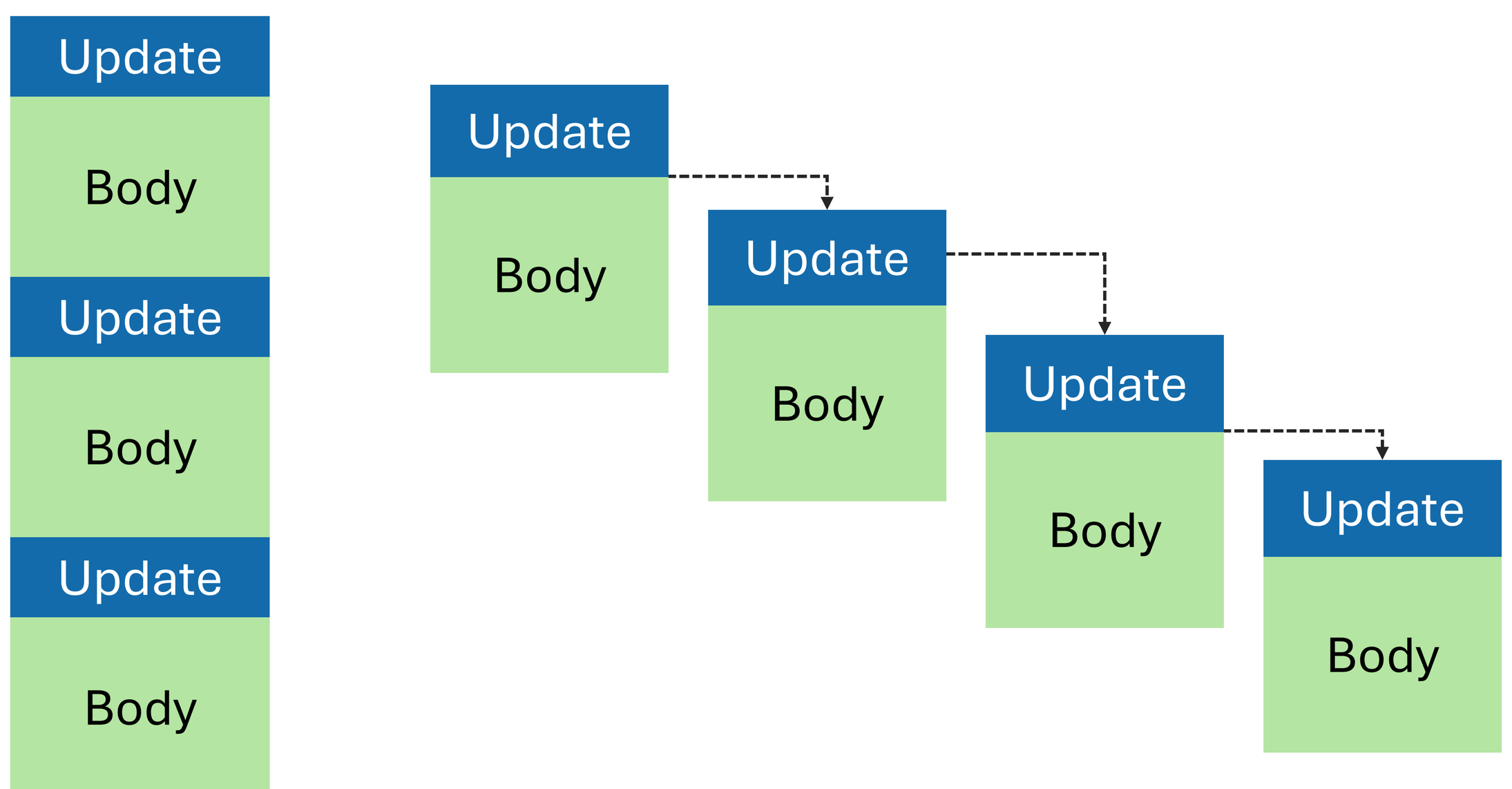


Figure 2.2: In program order, the sections appear sequentially (left). As ever, out-of-order execution is allowed, so long as sequential semantics are maintained. Thus, the core may parallelise (right).

III. Conflicts: squashing

The compiler is allowed to guess and be wrong.

Hardware verifies independence of register and memory operations, hides out-of-order speculation, and *squashes* if it detects a conflict.

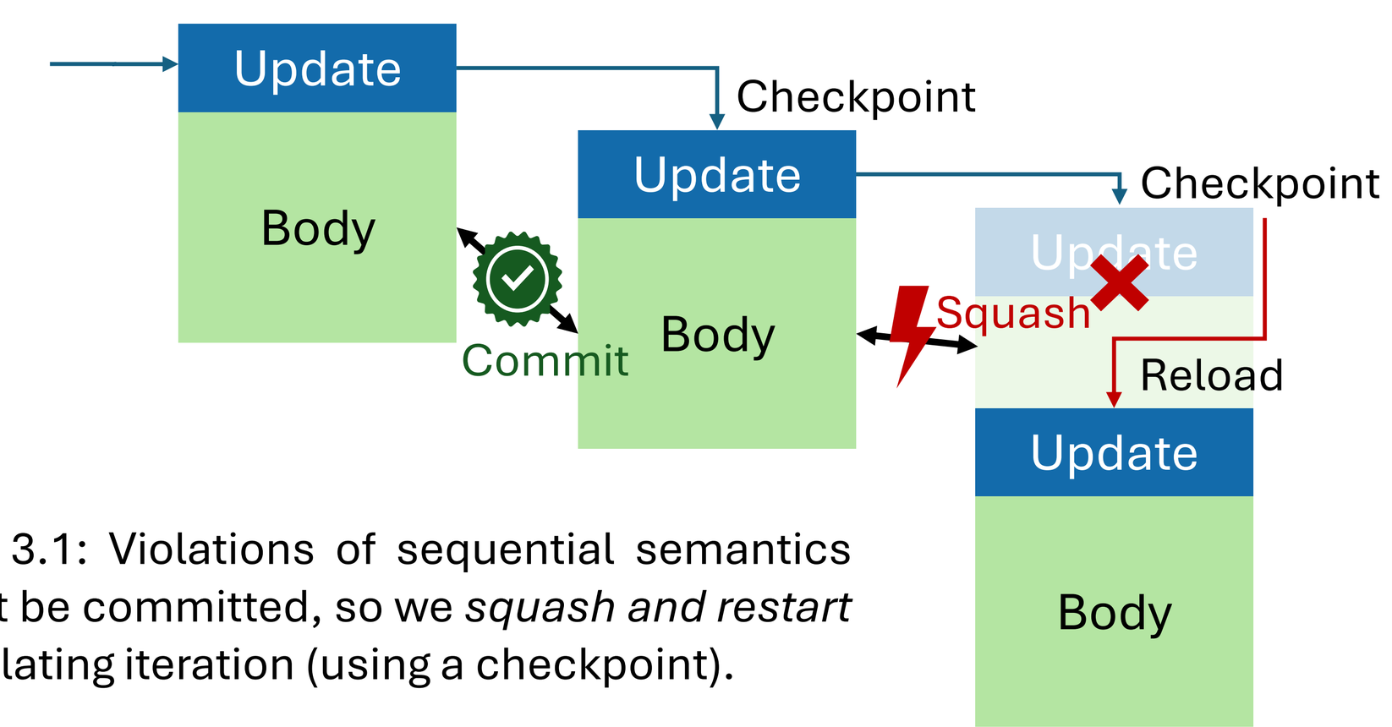


Figure 3.1: Violations of sequential semantics cannot be committed, so we *squash and restart* the violating iteration (using a checkpoint).

IV. Compatibility: crucial

Modern high-performance CPU cores are incredibly complex, with technology developed over decades. Thus, compatibility is key for obtaining real gains.

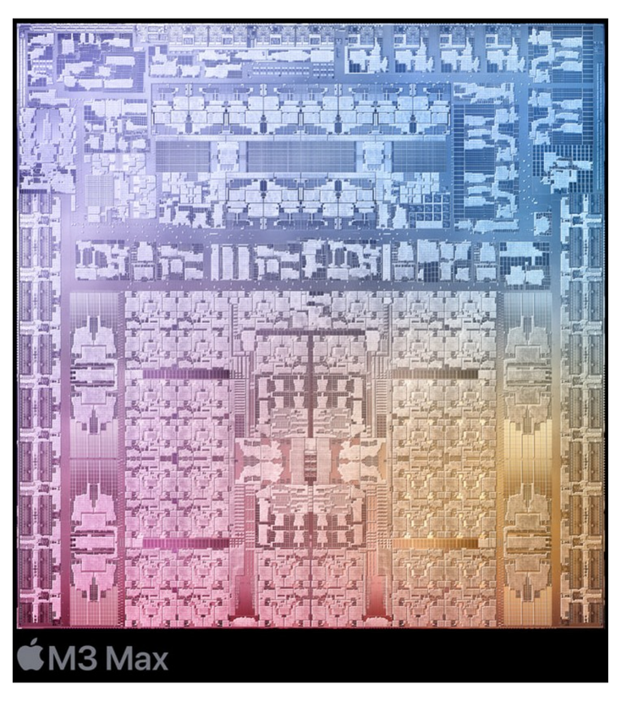


Figure 4.1: The Apple M3® Max chip. Modern high-performance processors are highly optimised and complex.
Source: Apple.com

We achieve this in four ways:

- Maintain ***sequential semantics*** (locally).
- ***Coherence:*** constrain speculation *to a single core.*
- ***Consistency:*** expose operations *in order*, or *atomically.*
- ***Limit modifications*** to the pipeline, ISA and programming model.

V. Microarchitecture

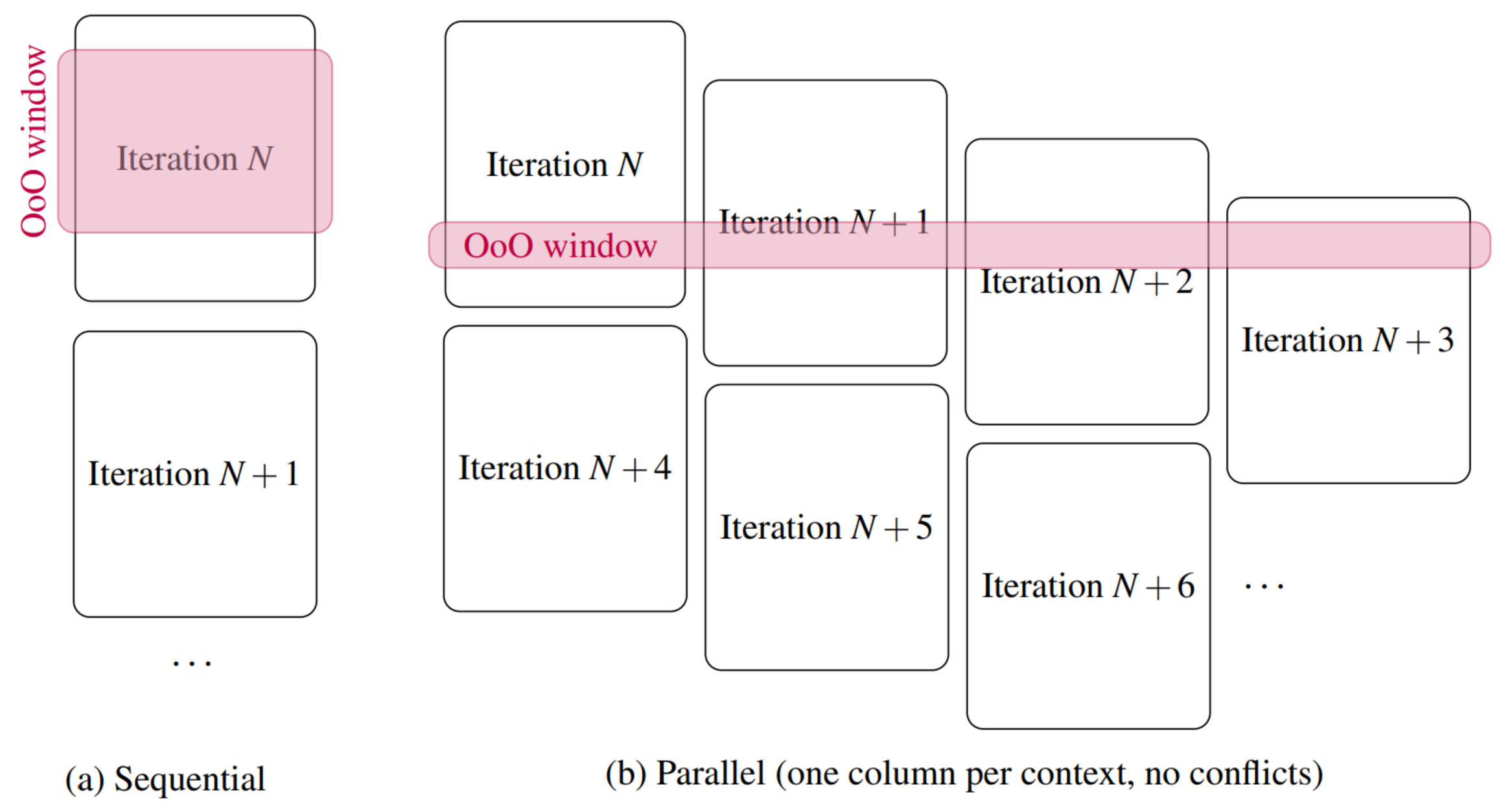


Figure 5.1: We add execution contexts to the pipeline. Each context runs one iteration, on an independent slice of the out-of-order window. This provides abundant instruction-level parallelism, which increases pipeline utilisation.

Pipeline resources are shared dynamically between multiple *execution contexts,* like in simultaneous multithreading (SMT).

Each context has its own

- slice of the out-of-order window,
- control state,
- architectural registers, and
- place in program order (totally ordered).

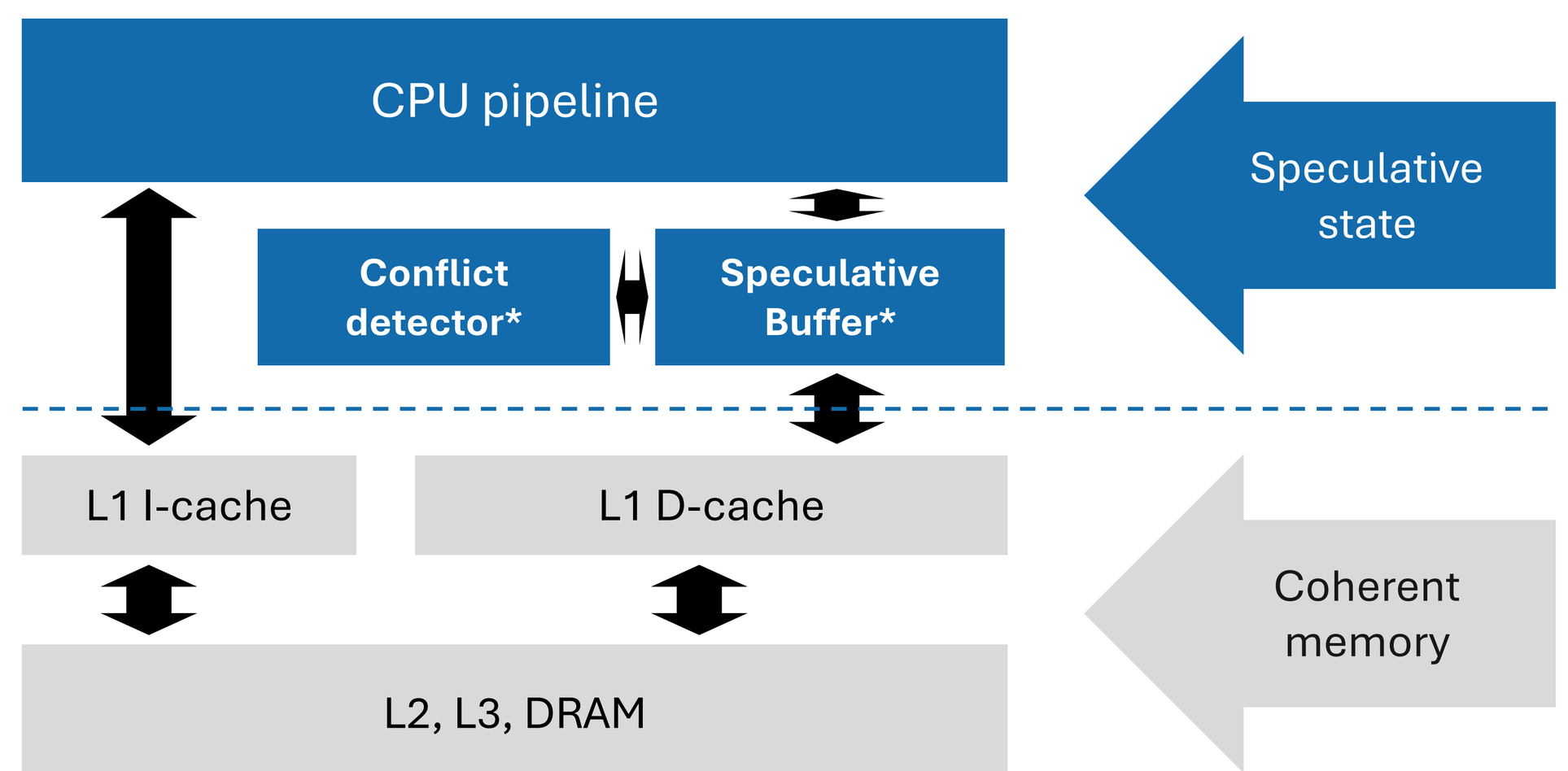


Figure 5.2: We add a *speculative buffer* between the CPU and the L1 data cache to buffer, hide and atomically commit speculation, and a *conflict detector* to check for violations. (*=new component, blue=exposed to speculation).

The oldest context runs **architecturally**, reading from & writing to the main memory system directly.

Other contexts are **speculative** and their writes are buffered. These can be restarted or discarded for any reason.

The **conflict detector** checks for conflicts *between* contexts, preserving sequential semantics *locally*.

The **speculative buffer**

- Eliminates write-after-write (WaW) and write-after-read (WaR) hazards using multi-versioning,
- Detects coherence conflicts with *other cores* by acquiring cache lines and snooping coherence traffic, and
- Enables *atomic* commit of contexts *in program order.*

VI. Iteration packing

Parallelisation must improve utilisation to get a speedup. However, short contexts cannot effectively fill the re-order buffer (ROB), as shown in Fig. 6.1.

We mitigate this by packing multiple iterations into each context. We predict updates to facilitate the jump-ahead.

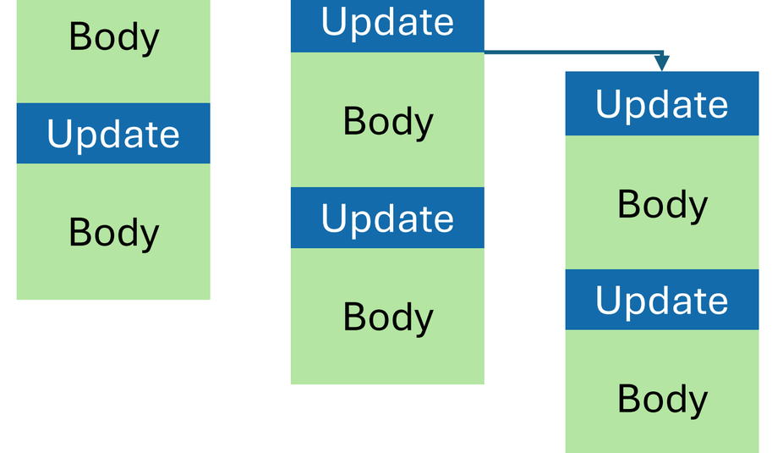


Figure 6.2: Parallel loop with a packing factor of 2 iterations per context. Updates "in the middle" of contexts are bypassed using value prediction.

Size predictor	≈ 234 instructions per iteration
Induction variable predictor	r10, r12
Value predictor (iteration i)	$r10 = 4 \times i + 465282$, *confident*
	$r12 = -1 \times i + 1024$, *confident*

Figure 6.3: We predict updates using 3 predictors:
Size predictor: suggests a suitable packing factor.
IV predictor: predicts induction variable registers.
Value predictor: predicts IV registers (stride).
Packing is performed if all predictors are confident and predicted iteration size is low.

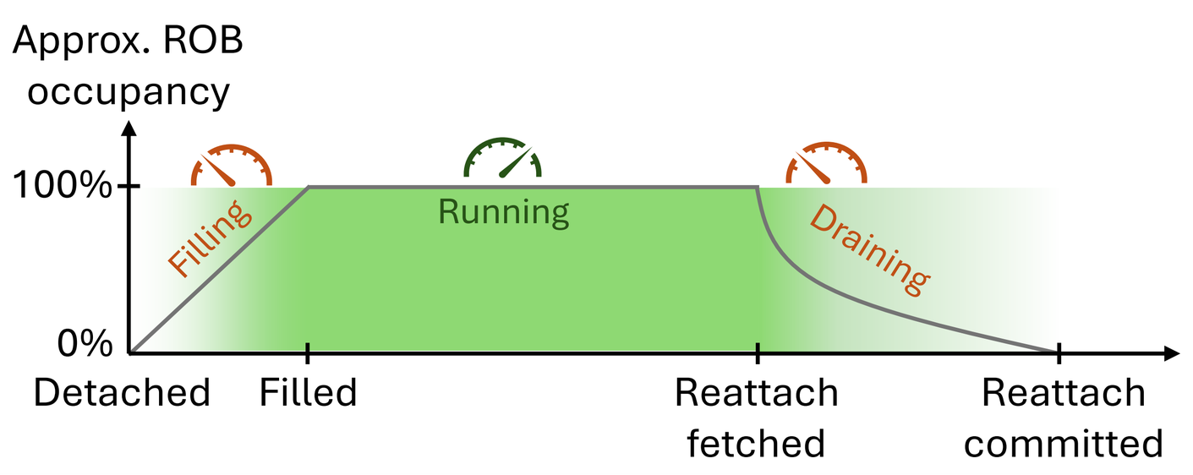


Figure 6.1: Performance depends on high ROB occupancy. This is low after startup (filling) and before shutdown (draining). If the context is long, the middle (running) phase dominates, otherwise performance is poor.

VII. Results

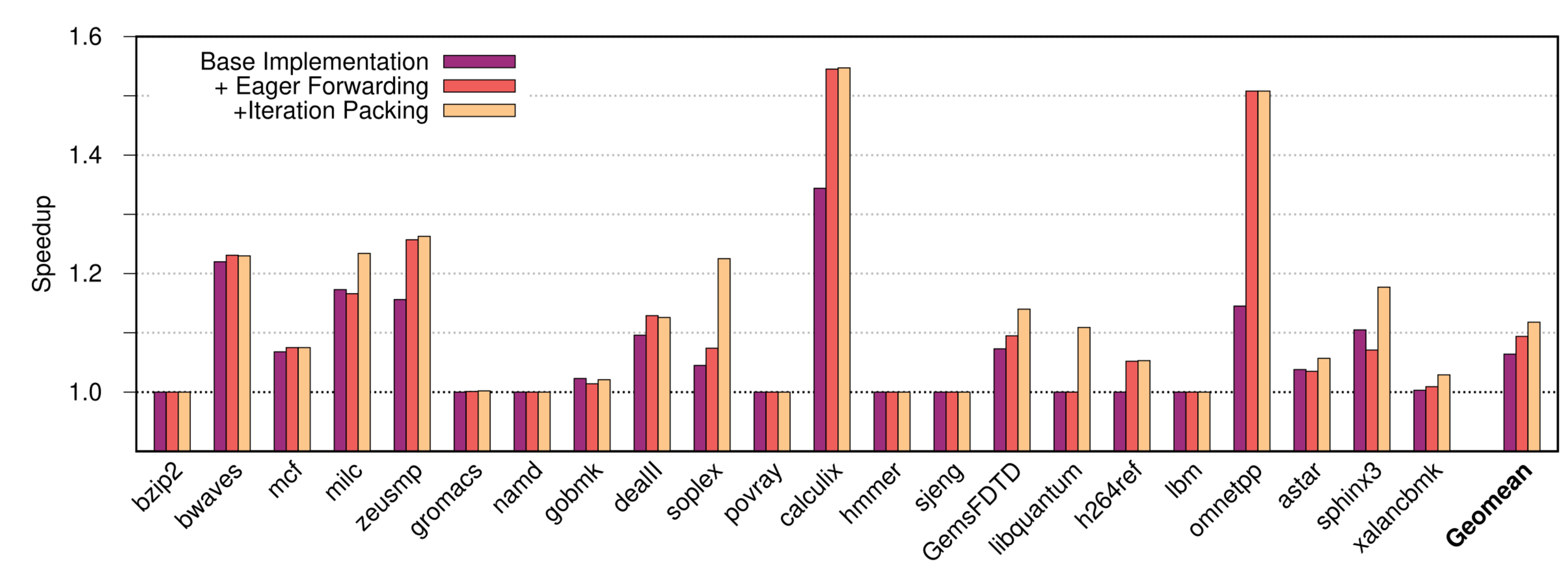


Figure 7.1: Speedup over an 8-wide out-of-order baseline core in Gem5 for SPEC CPU2006 benchmarks using Simpoints with optimisations added in one-by-one. The base implementation achieves a geometric mean speedup of **6.4%**, with eager forwarding adding **3.0%** and iteration packing a further **2.4%**, for a total of **11.8%**.

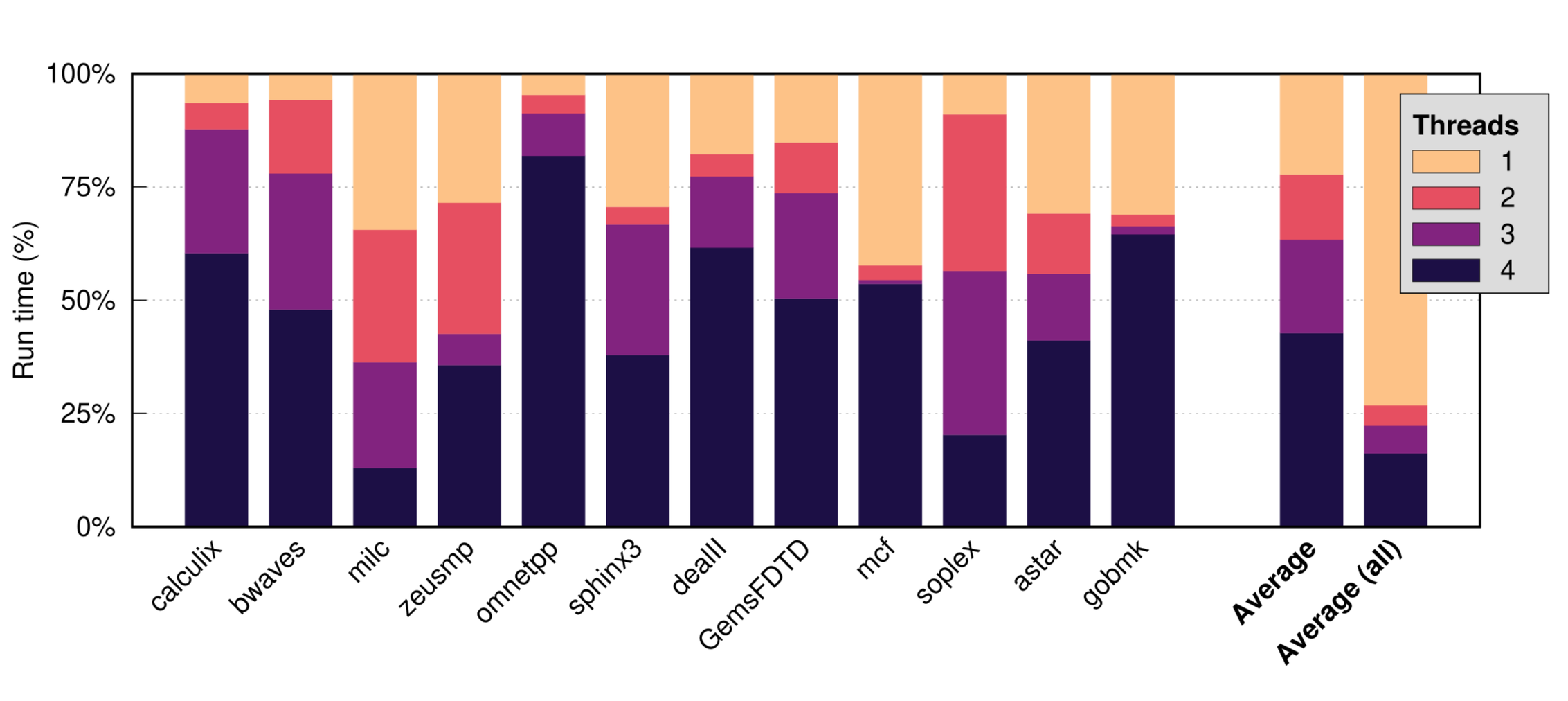


Figure 7.2: Number of contexts active by percentage of run time in SPEC CPU2006 benchmarks (base impl.). Many loops are unprofitable, thus coverage is only **27%**.

We implement a detailed prototype in the **Gem5 microarchitectural simulator**, and a prototype compiler in **LLVM** (see other poster) that can automatically transform loops and insert hints.

We achieve some promising results, although future work remains to parallelise more loops well.

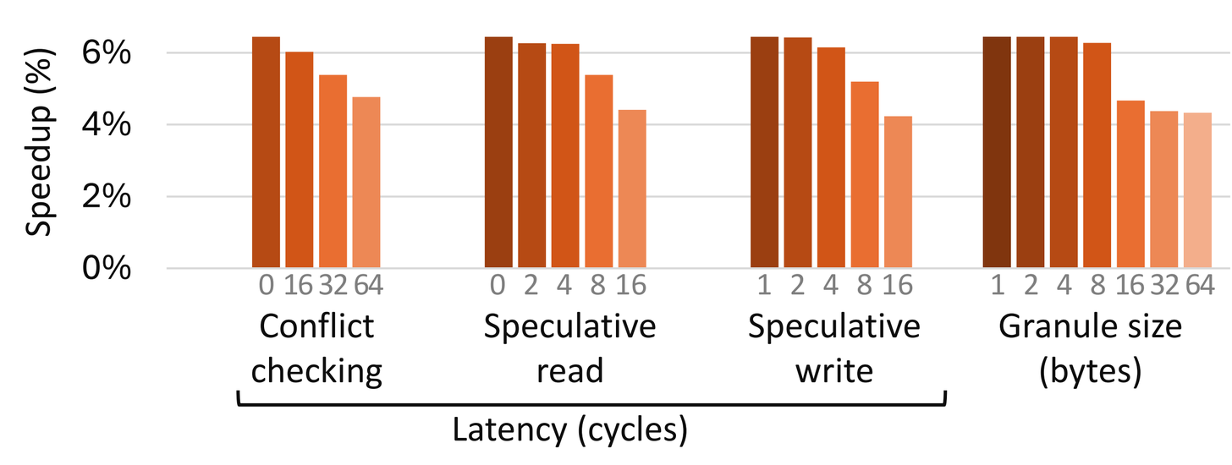


Figure 7.3: Results are stable with respect to small changes in latency and granule size, but larger changes hurt speedups.

We would like to thank ARM Ltd. and EPSRC (grant EP/W00576X/1) for generously funding our research.