An Empirical Framework for Identifying Instruction Fusion Candidates

Michael Shires
University of Virginia
jrs6qe@virginia.edu

***Abstract*—Instruction fusion has delivered sustained single-core performance gains, from x86 macro-op fusion of compare-and-branch, to RISC-V macro-op proposals, to recent Intel Advanced Performance Extensions (APX). Yet the choice of *which* pairs or chains to fuse has largely been driven by a small set of hand-picked patterns. We propose an empirical framework that systematically identifies high-impact fusion candidates using three complementary profiling strategies: (i) naive consecutive-pair frequency, (ii) backslices of last-level cache (LLC) misses, and (iii) dependency chains observed when the reorder buffer (ROB) stalls behind a pending LLC miss. Recent work on non-consecutive and compiler-assisted fusion shows that the practical fusion window extends well beyond adjacent pairs, dramatically enlarging the candidate space. Using SPEC CPU 2017 we show each strategy surfaces qualitatively different candidates, and we use Intel APX as a case study to contrast ISA extension with late-stage microarchitectural fusion.**

I. Introduction and Motivation

Although heterogeneous and parallel designs dominate much of the recent literature, single-core performance remains the critical bottleneck for a large class of existing and emerging workloads, such as control-heavy codes, interpreters, and coding agents. Instruction fusion has been a reliable lever for improving this performance without growing the architectural register file or inflating the ISA: fusing CMP+Jcc into a single operation, fusing ADD+store-address, or fusing a compressed shift-and-add idiom in RISC-V all reduce work-per-retired-instruction without perturbing the programming model [1], [8], [9].

APX introduces new data destination (NDD) forms of legacy integer instructions, conditional compares/tests, and zero-upper variants [2]. Each of these is, in effect, a fusion idiom promoted into the ISA: an implicit `MOV` fused into the producer, a `CMP` fused with a conditional update, a `MOV` fused with a zero-extension. Every such addition consumes opcode space, validation effort, and silicon, and is permanent. Given that recent non-consecutive fusion proposals [3] and compiler-assisted fusion [1], [10] show that *microarchitectural* fusion can recover much of the same benefit, our research question is: *can the fusion candidates that would matter most be identified empirically, before the ISA or silicon commits to them?*

Contributions. This paper sketches an empirical framework that answers that question with three profiling strategies, specifically consecutive-pair frequency, LLC-miss backslices, and ROB-saturation dependency chains—evaluated on SPEC CPU 2017, with Intel APX used as a case study.

II. Background and Related Work

Macro- and micro-op fusion. Intel has fused decoded macro-ops since Core and micro-fused load-ops since Sandy Bridge [8], [9]. Celio et al. [1] showed that RV64GC with macro-op fusion recovers 5.4% of the dynamic instruction gap versus x86-64 on SPEC CINT2006. Recent RISC-V implementations on XiangShan propose additional fusion pairs motivated by the B extension [10]. These works concentrate on *consecutive, contiguous* fusion.

Non-consecutive fusion. Helios [3] demonstrates that roughly 5.6% of dynamic memory μ-ops belong to non-consecutive, non-contiguous fusion pairs, and presents a microarchitecture to fuse across intervening instructions via allocation-queue inspection and predictive fusion. We plan to quantitatively compare against Helios as a baseline for the candidate space recoverable beyond adjacent-pair profiling.

Compiler-assisted fusion. Celio et al. observe that much of the latent RISC-V fusion potential is only unlocked when the compiler is aware of the target's fusion idioms [1]. Together with non-consecutive fusion, this closes the loop: if the compiler can legally reorder dependent instructions into the same basic block or ROB window, the microarchitecture can fuse them even without literal adjacency.

Backslice analysis and dependent misses. Zilles and Sohi [4] characterized backward slices of performance-degrading instructions (L2 misses, mispredicts) and showed most have small, pre-executable slices. Hashemi et al. [5] migrate slices of dependent cache-miss chains to an enhanced memory controller. These works established that cache-miss-terminated dependency chains are both identifiable and responsible for a disproportionate share of stall cycles—exactly the property that makes them attractive fusion targets in our setting.

III. Proposed Methodology

We profile SPEC CPU 2017 workloads using the simpoint and pinball methodologies, and then run the pinballs through the Sniper simulator, which has been modified to obtain our desired data.

We run the entire workload first, obtaining a deterministic replay whole-program pinball. Second, we replay that pinball, profiling and obtaining basic block vectors. Third, we group the basic block vectors, obtaining representative regions. Fourth, we replay the whole-program pinball and record simpoint pinballs for each region. Finally, we can replay the simpoint pinballs within Sniper, obtained our necessary data.

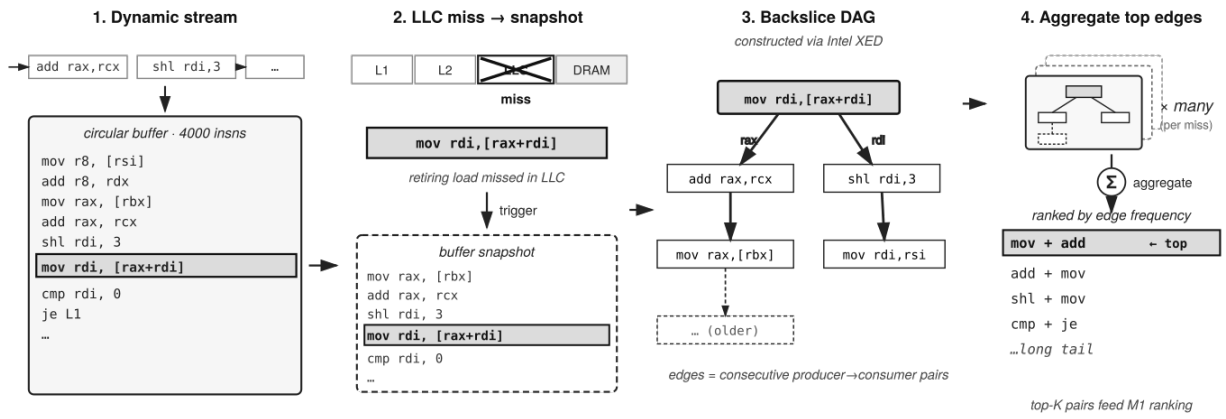

Fig. 1. LLC-miss backslice workflow (M2).

(M1) Consecutive pair frequency. The baseline: a histogram of consecutive pairs over the dynamic trace, filtered by register-dependency. This replicates Celio et al.'s and Intel's idiom-matching approach, and provides a starting point for most commonly executed pairs of instructions.

(M2) LLC-miss backslices. While the simpoint is running, we keep a circular buffer of 4000 most-recently executed instructions. When a retired load misses in the LLC, we take a snapshot of the buffer and use Intel XED to create dependency tree of instructions leading up to the LLC miss, or "backslice". Each edge on this tree represents a consecutive instruction pair that must be executed prior to issuing the LLC miss. Pairs that appear disproportionately often on the critical path to LLC misses are high-value candidates, as fusing them contracts the chain and issues the miss sooner [7].

(M3) ROB-saturation chains. M2 ignores whether the miss-to-be is actually *blocked* by dispatch. In M3 we will sample cycles in which the ROB is full and the next instruction waiting to dispatch is a load predicted to miss in the LLC. At those cycles we inspect the in-flight instructions in the ROB and extract the subchain leading to the stalled load. Candidates here are weighted by the stall cycles they cost, not by raw frequency.

System aspects affected. Candidate selection itself does not affect an architecture system, however the process of implementing fusion in novel architectures can impact the processor front-end, micro-op cache, ROB, and execution via changes in functional units and ports available.

IV. Implementation Pathways

While candidate *identification* is the primary contribution, we plan to study two implementation routes so candidate rankings can be weighted by realizable benefit:

(P1) Fused encoding in the μop cache. When a known fusion pair is decoded, the μop cache stores a single fused slot, saving decode bandwidth, μop-cache capacity, and a rename port. This generalizes the compressed form Intel already uses for CMP+Jcc and can capture non-consecutive pairs when the intervening μops are independent, following Helios [3]. We are also planning on exploring a "scratchpad" of encoding space, where the processor can dynamically determine fusion pairs at run time, and assign them in the scratchpad to an available encoding value, and use this new fused μop to save pressure in the μop cache.

(P2) New execution ports. A functional unit with a new execution port designed to execute the fused pair in a single issue slot. This path is more expensive but captures pairs where both halves are data-producing (e.g., dependent integer arithmetic) rather than the compare-and-branch style P1 handles best.

We expect M2 and M3 to favor P2 (dependent producer chains on the critical path) while M1 will favor P1 (frequent, simple idioms already close to Intel's macro-fusion family).

V. Evaluation Plan

Workloads. SPEC CPU 2017, Integer and floating point workloads. We have completed backslice extraction over all benchmarks and observe meaningful inter-benchmark variance in the top candidate pairs. We have created an automated workflow to produce this data from a binary and arguments, so we plan on extending this to additional workloads.

Tools. We use Intel SDE to produce pinballs, Simpoint to group simulation slices into regions, Intel SDE again to produce simpoint pinballs, and Sniper to perform the simulation. We use Intel XED within Sniper to generate producer-consumer dependency trees.

Metrics. The reduction in Dynamic-instruction-count will be the primary metric, as well as the comparison of IPC, memory-level parallelism (MLP) in memory-bound benchmarks, and μop-cache hit rate.

Comparisons. For each workload, we have compiled it with and without APX support. This allows us to compare how the fusion candidates change with the new instruction extension, and evaluate the extension's performance against a baseline. We use LLVM as our compiler, and have already identified one segmentation-fault issue within LLVM when compiling with APX support, which has been reported and fixed.

VI. Conclusion

Intel APX, along with modern RISC-V extensions, shows the field is still actively promoting fusion candidates into new instructions. The converse design point, that identifying candidates empirically and fusing them at run-time, has mostly been explored one candidate at a time. We propose a framework that combines naive, LLC-miss-guided, and ROB-pressure-guided profiling to empirically identify the most compelling fusion candidates, and two methods of implementing these design choices, which have shown to exhibit performance gains in prior work. We hope workshop feedback will help shape the evaluation plan before a full submission.

References

[1] C. Celio, D. Dabbelt, D. A. Patterson, and K. Asanović, "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V," *Tech. Rep. UCB/EECS-2016-130*, July 2016.

[2] Intel Corporation, "Intel® Advanced Performance Extensions (Intel® APX) Architecture Specification," Document 355828, Rev. 3.0, 2024.

[3] S. Singh, A. Perais, A. Jimborean, and A. Ros, "Exploring Instruction Fusion Opportunities in General Purpose Processors," in *Proc. MICRO-55*, 2022, pp. 1–13.

[4] C. B. Zilles and G. S. Sohi, "Understanding the Backward Slices of Performance Degrading Instructions," in *Proc. ISCA-27*, 2000, pp. 172–181.

[5] M. Hashemi, Khubaib, E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Accelerating Dependent Cache Misses with an Enhanced Memory Controller," in *Proc. ISCA-43*, 2016, pp. 444–455.

[6] I. Kim and M. H. Lipasti, "Macro-op Scheduling: Relaxing Scheduling Loop Constraints," in *Proc. MICRO-36*, 2003, pp. 277–288.

[7] K. Kokolis, P. Moreau, A. Perais, and J. Torrellas, "Dependence-aware Slice Execution to Boost MLP in Slice-out-of-order Cores," *ACM TACO*, vol. 19, no. 2, 2022.

[8] N. Hinton et al., "System and Method for Fusing Instructions," US Patent 6,675,376, 2004.

[9] WikiChip, "Macro-Operation Fusion (MOP Fusion)," https://en.wikichip.org/wiki/macro-operation_fusion.

[10] Y. Yao et al., "Implementing the RISC-V B Extension with Instruction Fusion on the XiangShan Processor," in *Proc. IEEE/ACM Conf.*, 2023.

[11] GNU Project, "GCC Internals: Scheduling (`TARGET_SCHED_MACRO_FUSION_P`, `TARGET_SCHED_FUSION_PRIORITY`)," https://gcc.gnu.org/onlinedocs/gccint/Scheduling.html.

[12] N. Gober et al., "The Championship Simulator: Architectural Simulation for Education and Competition," arXiv:2210.14324, 2022.