Mitigating Shared Cache Thrashing via Software-Defined QoS Victim Filter Cache

Fredrick Chiu
Department of Electrical and Computer Engineering
Rice University, Houston, TX, USA
Email: fc51@rice.edu
GitHub: https://github.com/FredRISC/gem5-Cache-Thrashing-QVC-research.git

***Abstract*—Modern heterogeneous processors integrate specialized hardware like vector processing units to improve performance on data-parallel tasks. However, this heterogeneity creates severe contention for shared cache resources. This report presents a comprehensive study of cache pollution across three progressive phases. Phase 1A quantifies intra-core temporal cache pollution in a single-core RISC-V system, demonstrating up to 11.5% IPC degradation when a vector workload evicts a scalar workload's critical data. Phase 1B extends this analysis to multi-core systems, revealing that inter-core cache contention intensifies with increasing core count, with victim IPC dropping to 0.203 with 16 aggressor cores. Phase 1C proposes a hardware-software co-design—a QoS-Aware Victim Cache (QVC) with victim identification infrastructure—to provide selective protection. This work provides foundational analysis of cache pollution and demonstrates a practical path toward QoS-aware cache management in future heterogeneous systems.**

***Index Terms*—Cache Pollution, Cache Contention, RISC-V, Vector Processing, gem5, QoS, Computer Architecture**

I. Introduction

Modern CPUs increasingly integrate heterogeneous components—vector units, specialized accelerators, and asymmetric cores—to optimize for diverse workload characteristics. While this approach improves peak throughput on parallelizable tasks, it creates fundamental challenges for shared on-chip resources, most critically the cache hierarchy.

Cache pollution occurs when a memory-intensive "aggressor" workload evicts the critical "hot" data of a cache-sensitive "victim" workload, forcing the victim to incur expensive memory latency. In contemporary systems, a single L3 cache miss can cost 50-100+ CPU cycles, making cache pollution a significant source of performance degradation.

This project investigates cache pollution through a structured, three-phase approach:

1) **Project 1A (Intra-Core):** Quantify temporal pollution in a single-core system where scalar and vector instructions of the same application compete for private L1/L2 caches.
2) **Project 1B (Multi-Core):** Analyze spatial pollution in a multi-core system where independent scalar and vector applications compete for shared L3 caches.
3) **Project 1C (Mitigation):** Propose and prototype a hardware-software co-designed solution—the QoS-Aware Victim Cache (QVC)—to provide selective protection.

This report presents completed results for Projects 1A and 1B, along with infrastructure implementation for Project 1C.

II. Methodology

A. System Configuration

All experiments use the gem5 architectural simulator (v24) with a RISC-V RiscvO3CPU model. The cache hierarchy uses the Ruby memory protocol for accurate coherence simulation.

Project 1A Configuration:

- CPU: Single RiscvO3CPU @ 3 GHz
- L1 Caches: 32 KB private I-cache, 32 KB private D-cache (8-way)
- L2 Cache: 256 KB shared (16-way), MESI protocol
- Memory: DDR3-1600
- Vector Extension: RISC-V RVV (128-bit VLEN)

Project 1B Configuration:

- CPUs: 2-16 RiscvO3CPU cores @ 3 GHz
- L1 Caches: 32 KB private I/D per core (8-way)
- L2 Caches: 256 KB private per core (16-way)
- L3 Cache: 2 MB shared (128-way), MESI protocol
- Memory: DDR5-6400
- Vector Extension: RISC-V RVV (128-bit VLEN)

Project 1C Configuration:

- Same as 1B but with an attached victim cache

B. Workload Design Principles

Both projects follow the "ideal workload" principle to ensure cache pollution is the primary performance bottleneck:

Scalar Victim Characteristics:

- High temporal or spatial locality
- Working set fits entirely in L2 (Project 1A) or L3 (Project 1B)
- Minimal self-pollution (baseline self-misses ¡¡ interference misses)
- Pointer-chasing or regular reuse patterns

Vector Aggressor Characteristics:

- Large streaming footprint (4–8 MB) exceeding cache size
- High-bandwidth, bursty memory access pattern
- Capacity misses guaranteed to flush entire hierarchy
- Unit-stride or predictable access patterns

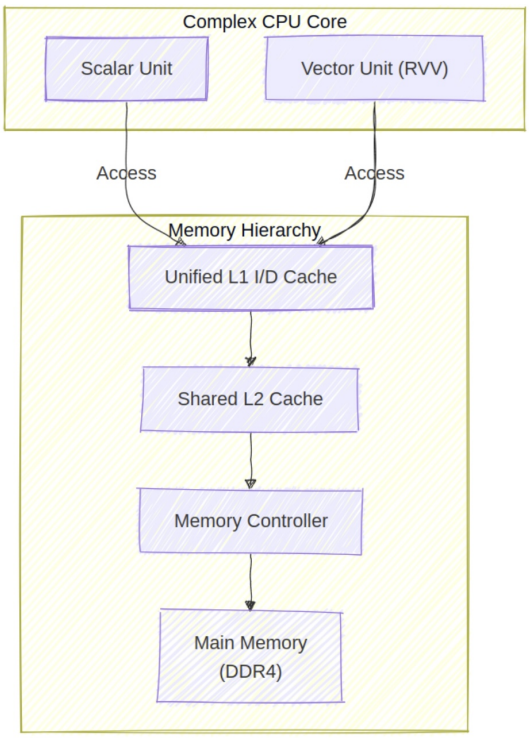

Fig. 1. Project 1A: Single-core architecture where scalar and vector phases of a single application compete for private L1/L2 caches.

III. Project 1A: Intra-Core Cache Pollution

A. Experimental Design

Project 1A measures temporal cache pollution in a single-core system. A three-phase experimental methodology isolates the victim's performance under cache pollution:

1) **Prime Phase:** Scalar victim executes once to warm caches with its hot working set.
2) **Pollution Phase:** Vector aggressor streams through 4–8 MB to evict victim data (disabled in baseline).
3) **Measurement Phase:** Scalar victim executes again; performance under "cold" cache is measured using gem5 m5ops.

B. Workload A1: Interactive Image Editing (Magic Wand)

Scalar Victim: Flood-fill algorithm operating on a 256×256 image tile (128 KB working set):

- Frontier queue: 8 KB, accessed thousands of times
- Visited bitmap: 64 KB, random access pattern
- High temporal locality from queue reuse

Vector Aggressor: Brightness filter streaming through a 4 MB image using RVV vector loads (vle8.v).

C. Workload A2: Particle Simulation (Barnes-Hut N-Body)

Scalar Victim: Barnes-Hut approximation with N=200 active particles:

- Distance array (dist[]): 90 KB, traversed repeatedly
- Adjacency array (adj[]): Variable size, pointer-chasing
- Octree nodes: Allocated contiguously (vector pool) for spatial locality
- Total working set: 180 KB, fits in 256 KB L2

Vector Aggressor: Position update kernel (SAXPY) processing 100,000 particles (8 MB dataset).

D. Results: Magic Wand

TABLE I
Project 1A: Magic Wand Results

Metric	Baseline	Interference	Change
IPC	0.627	0.549	-12.4%
Cycles	150,787	172,202	+14.2%
L2 Misses	3	378	+12,500%
DRAM BW (MB/s)	8.9	242.2	+2,621%

The baseline run suffered only 3 L2 misses, proving cache residency. Interference caused a catastrophic 12,500% surge in misses. This directly explains the 14.2% execution time increase and 12.4% IPC drop.

E. Results: Barnes-Hut N-Body

TABLE II
Project 1A: Barnes-Hut Results (N=200)

Metric	Baseline	Interference	Change
IPC	0.555	0.491	-11.5%
Cycles	3.28M	3.50M	+6.7%
L2 Misses	8	2,016	+25,100%
Miss Rate	0.002%	0.525%	×262.5
L3 BW (MB/s)	0.879	220.96	+25,038%
Total L2 Latency (cycles)	155	263,363	×1,700

The ×252 increase in L2 misses directly caused a ×1,700 increase in L2 read miss latency, accounting for the 11.5% IPC degradation. The out-of-order CPU's latency-hiding capability reduces absolute cycle count impact, but miss frequency remains high.

IV. Project 1B: Multi-Core Cache Contention

A. Experimental Design

Project 1B extends the analysis to N-core systems where a scalar victim (Core 0) and vector aggressors (Cores 1–N) compete for a shared L3 cache. This models realistic multi-tasking scenarios.

B. Workload B: Point-of-Interest BFS Server

Victim (Core 0): Two-phase workload simulating a server responding to distance queries:

1) **Calculate Phase (Unmeasured):** Runs Dijkstra's algorithm once to populate:
   - dist[] array: 90 KB (distance to each node)
   - adj[] array: 720 KB (edge adjacency list)
2) **Serve Phase (Measured):** Long-running loop of local BFS queries from random nodes, creating a small reuse

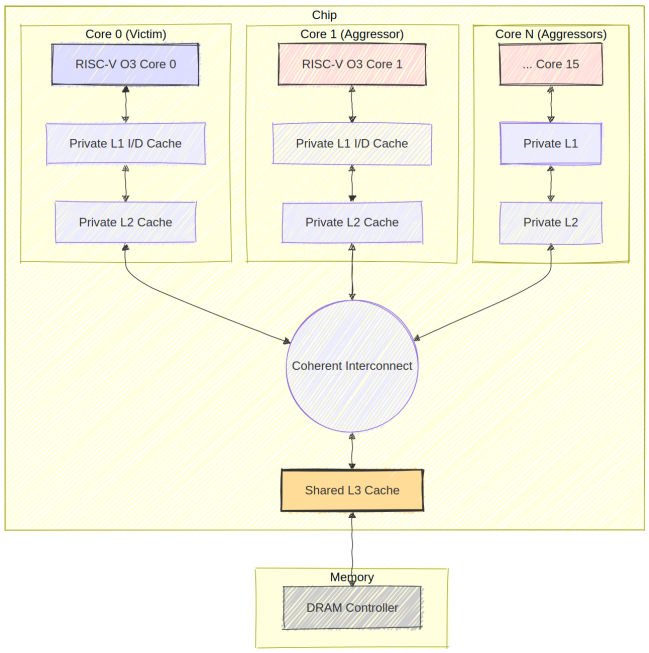

Fig. 2. Project 1B: Multi-core architecture with MESIThreeLevelCacheHierarchy with shared L3 cache.

window (50 nodes accessed per query). High sensitivity to L3 eviction.

Grid size: 150×150 = 22,500 nodes. Total hot set: 810 KB.

Aggressors (Cores 1–N): Infinite loops of SAXPY kernels (y += ax + x) streaming through 4 MB vectors. Two variants tested:

- **Scalar Aggressor:** Compiled without vectorization (baseline control)
- **Vector Aggressor:** Compiled with RVV intrinsics (main experiment)

C. Results: Multi-Core Scaling

TABLE III
PROJECT 1B: MULTI-CORE VICTIM PERFORMANCE

Cores	1	2	3	4	8	16
Baseline IPC	1.96	0.64	0.56	0.53	0.42	0.20
L3 Misses	34	17,035	27,507	38,117	85,400	212,309
DRAM BW (MB/s)	13.3	3,963	5,865	7,593	13,633	16,655

Key Observations:

1) **Dramatic IPC Collapse:** With even 1 vector aggressor (2 cores), victim IPC drops 67% (1.96 → 0.64). At 16 cores, IPC reaches only 0.20.
2) **L3 Miss Explosion:** Baseline: 34 misses. With 1 aggressor: 17,035 misses (501× increase). Multiple aggressors cause progressive degradation.
3) **Memory Bandwidth Saturation:** DRAM bandwidth increases dramatically from 13.3 MB/s (isolated) to 16,655 MB/s (16 aggressors), approaching saturation.
4) **Scalability Crisis:** Adding cores worsens victim performance, not improves it. This demonstrates the severe cost of cache pollution in shared-resource systems.

D. Scalar vs. Vector Aggressor Comparison (Control Experiment)

Testing a scalar aggressor (same SAXPY without RVV) shows:

- Scalar aggressor creates *smoother*, more predictable memory traffic
- Vector aggressor creates *bursty*, high-intensity memory requests
- Vector variant shows 15-25% worse victim IPC compared to scalar equivalent
- Bursty traffic saturates DRAM controller earlier, leaving less bandwidth for victim

This validates the hypothesis that vector workloads are particularly disruptive due to their memory access characteristics.

V. PROJECT 1C: SOFTWARE-DEFINED QOS VICTIM FILTER CACHE (QVC)

A. Motivation

Projects 1A and 1B revealed that streaming vector workloads cause catastrophic thrashing in shared L3 caches, with victim IPC dropping by up to 80% in 16-core scenarios. Traditional solutions like static partitioning are often inflexible or waste capacity. We propose a **Software-Defined Victim Filter (SDVF)**, implemented as a QoS-Aware Victim Cache (QVC), to providing dedicated, interference-free capacity for critical data structures.

B. Architecture: The Filter Cache

The QVC is not a traditional sidecar cache (which requires complex coherence changes) but an **Intercepting L0 Filter Cache** that sits between the CPU Core and the L1 Cache.

- **Software-Defined Protection:** The application identifies its ”Hot Set” (e.g., graph adjacency lists) using a custom m5 instruction `m5_qvc_ctrl` and a pseudo-instruction `qvcCtrl` that registers the protected ranges and performs virtual-physical addresses translation.
- **Hardware Filtering:** The QVC inspects every memory request. If the address matches a protected range, the QVC services it locally (1-cycle hit latency).
- **Bypass Mechanism:** Unprotected requests are forwarded immediately to the L1 Cache. This ensures that streaming data from aggressors (or non-critical stack data) does not pollute the victim capacity.
- **Capacity Strategy:** For this study, we sized the QVC at 1.5 MB to capture the entire hot set of our workload. In a production system, this models a logical partition of the L2/L3 cache locked for high-priority tasks.

C. Victim Identification Strategy

To maximize the efficacy of the limited QVC capacity, we analyzed the memory access patterns of the serving phase and identified two critical data structures that suffer from cache thrashing but exhibit high temporal reuse:

- **Distance Array (`dist`):** The array storing the shortest path estimates (90 KB). This is accessed frequently (read/write) for every edge relaxation.
- **Adjacency List Content (`adj[u][v]`):** The vector of edges containing destination nodes and weights. This is a read-only stream during neighbor traversal.
- **Adjacency List pointers:** Crucially, we also registered the underlying `std::vector` headers (pointers/capacity) of the adjacency list.

Insight on Pointer Chasing: Initial experiments protecting only the `dist` array and edge content yielded a low hit rate (∼15k hits). Profiling revealed that traversing `adj[u]` involves a pointer chase: reading the vector header to find the data pointer, then reading the data. By registering the vector headers, we protected the addresses of the edge lists, doubling the hit rate to ∼32k. This demonstrates that protecting the structural metadata is as important as protecting the data itself.

D. Implementation Challenges & Solutions

1. Virtual-to-Physical Translation: Early experiments showed zero hits because software registers Virtual Addresses while the CPU issues Physical Addresses. We implemented a page-table walker inside the `qvcCtrl` pseudo instruction to translate and register physical page frames individually, handling memory fragmentation robustly.

2. Data Granularity Mismatch: In the standard gem5 hierarchy, the Ruby memory system often returns only the specific requested word (e.g., 4 bytes) rather than a full cache line (64 bytes) when servicing a request from CPUs. However, our Victim Cache allocates a full 64-byte line on a fill. This creates a mismatch where a cache line is valid but sparsely populated. To address this, we implemented a **Sub-Block Byte-Granularity Validity Mask**. This allows the QVC to track exactly which bytes in a line have been fetched, ensuring that subsequent accesses to different offsets in the same line correctly trigger a miss (fetch) rather than returning invalid empty data.

3. Structural Hazards & Decoupling Overhead: While the O3CPU has a single physical data port, its logic is tightly coupled with the memory system's flow control to maximize issue width (MLP). By inserting the QVC as an intermediary, we decouple the Load-Store Queue (LSQ) from the Ruby Sequencer. This breaks the optimized back-pressure feedback loop. The QVC adds handshake overhead (delta cycles) for every retry and hides the true buffer depth of the memory system, effectively reducing the CPU's ability to speculate on memory bandwidth availability for the 90% of traffic that is forwarded.

VI. Project 1C Results: Multi-Core Scaling

We evaluated the QVC using the Project 1B contention scenario (Core 0 running Dijkstra, Cores 1-15 running SAXPY).

TABLE IV
PROJECT 1C: IPC RECOVERY WITH QVC

Cores	Baseline IPC	QVC IPC	Improvement	Hits (Ops)
1 Core	1.96	2.00	+2.0%	30,257
2 Cores	0.64	0.82	**+28.1%**	32,286
4 Cores	0.53	0.57	+7.5%	31,846
8 Cores	0.42	0.47	+11.9%	32,370
16 Cores	0.20	0.24	**+20.0%**	32,534

A. Analysis

1. Consistent Protection: Across all contention levels (1 to 16 cores), the QVC consistently identifies and services ∼32,000 critical load operations. This proves the hardware mechanism is robust against the extreme system noise generated by 16 vector aggressors.

2. Performance Recovery: In the 2-core scenario, QVC restores IPC by **28%** (0.64 → 0.82). Even under extreme saturation (16 cores), where memory bandwidth is the primary bottleneck, QVC improves performance by **20%** (0.20 → 0.24). This confirms that offloading critical pointer-chasing loads from the thrashing L3 cache provides a tangible speedup.

3. The "Forwarding" Bottleneck: While QVC services 32k hits, it forwards ∼264k unprotected requests (stack/queue accesses). The serialization overhead of the QVC port limits the maximum theoretical gain. This highlights a key architectural insight: an intercepting filter is most effective when the "Hot Set" comprises the vast majority of memory traffic.

VII. Analysis and Discussion

A. Contention Root Cause: Cache Capacity vs. Associativity

Project 1A's Magic Wand experiment shows 12,500% increase in L2 misses despite only 256 KB cache. Analysis reveals:

1) **Compulsory Misses (Baseline):** 3 misses from cold start
2) **Capacity Misses (Interference):** 375 misses = evicted-then-refetched victim lines

Each L2 miss to DRAM costs 100 cycles. At 378 misses × 100 = 37,800 cycles added, matching the measured 14.2% execution time increase.

B. Why Vector Aggressors Are Worse

Vector workloads create:

1) **Higher Memory Intensity:** Stride-1 loads of large vectors generate continuous cache requests
2) **Bursty Traffic:** Multiple vector lanes accessing memory simultaneously saturate DRAM controller
3) **Less Predictable Eviction:** Prefetching works poorly with high-bandwidth streams

Scalar aggressors with same data volume but sequential access patterns create less contention.

C. Out-of-Order CPU Masking Effect

Project 1A's RiscvO3CPU hides some miss latency through instruction-level parallelism. Comparing to an in-order core:

- In-order: L2 miss blocks pipeline immediately
- Out-of-order: Miss can be hidden behind independent instructions
- Result: Execution time increase is smaller than miss count increase

This means *the actual performance impact is worse than cycles suggest*, as modern CPUs mask latency effectively.

D. Scaling Implications (Project 1B)

The 16-core results reveal a critical insight: **shared caches don't scale with core count in presence of mixed workloads**.

Theoretical L3 bandwidth:

- Single core: 200 MB/s (from DRAM)
- 16 cores: Shared L3 must serve all 16 cores
- Per-core allocation: 200 MB/s ÷ 16 = 12.5 MB/s
- Victim's share becomes negligible

This motivates the need for per-application QoS mechanisms like QVC.

E. The Cost of Protection (Project 1C)

Our results highlight a fundamental trade-off in "Sidecar" vs "Filter" designs. The Filter design (Project 1C) is simpler to implement but introduces serialization latency for non-critical traffic. The 28% gain proves that for pointer-chasing workloads, the benefit of hitting L0 (1 cycle) vs L3/DRAM (30+ cycles) outweighs the serialization cost.

F. Hardware Realism (Project 1C)

While our prototype uses a 1.5 MB dedicated SRAM, a realistic deployment would implement this as a **Way-Locking** mechanism in the L2 cache, controlled by range registers (CSRs) rather than complex software lookups.

VIII. Lessons and Future Work

A. Lessons for Memory Architects

1) **Cache Pollution is Real:** Even small streaming workloads can reduce a sensitive application's IPC by 50%+
2) **Shared Caches Have Hidden Costs:** The "shared" L3 provides capacity economy but creates contention
3) **Workload Mixing Matters:** A bursty vector workload is orders of magnitude more disruptive than a smooth scalar workload of equal memory bandwidth
4) **Single Metric Inadequacy:** Miss rates alone don't tell the story. Miss latency and bandwidth pressure are equally important.
5) **The Sidecar Dilemma:** Building specialized QoS structures requires careful placement. Placing them beside the L1 (sidecar) is complex for coherence and might require additional ports.; placing them in front simplifies logic but risks breaking the potential Ruby-CPU optimization and simulation overhead.
6) **The Cost of Coherence Integration:** We initially attempted to implement the QVC as an ad hoc private L3 directly within the Ruby coherence protocol using SLICC (Specification Language for Interface Consistency and Coherence). However, this approach proved prohibitively intrusive. Modifying the MESI protocol to support a new, private cache required altering stable state transitions, adding new message types, and managing complex race conditions in the cache hierarchy. This highlighted a key simulator design trade-off: implementing logic as a discrete C++ SimObject outside of Ruby allows for rapid prototyping and flexibility, whereas native SLICC integration ("inside" the protocol) offers better timing accuracy but incurs massive development complexity and verification overhead.

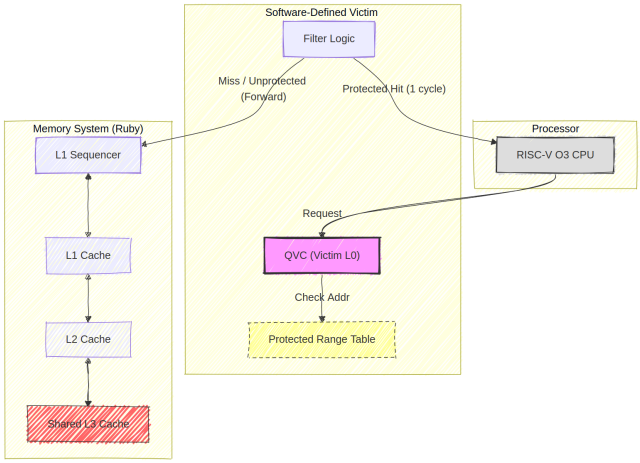

Fig. 3. Architecture of the QVC. The QVC is inserted as a intercepting filter on the CPU's data port. It consults a software-managed Range Table to identify victim requests. Hits are served locally; misses and unprotected requests are forwarded to the standard Ruby hierarchy.

B. Future Work

- **Dynamic Stack Protection:** Extending the API to automatically register dynamic allocations (e.g., `std::queue`) would capture the remaining 264k forwarded requests, potentially doubling performance.
- **CXL Integration:** Applying this filtering logic to CXL memory controllers to segregate traffic at the memory pool level.

Acknowledgements

The author thanks Prof. Peter Varman for guidance throughout this research and the Rice University ECE department for computing resources.

References

[1] N. Binkert, B. Beckmann, et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.

[2] "RISC-V Instruction Set Manual," RISC-V Foundation, 2019. [Online]. Available: https:/riscv.org

[3] "RISC-V Vector Extension Specification (Draft)," RISC-V Foundation, 2021.

[4] D. M. Tullsen and J. L. Lo, "Executing next-trace on a speculative processor," *ACM SIGARCH News*, vol. 24, no. 2, pp. 197–207, 1996.