Cache(d) Compression: Increasing Effective Storage within RISC -V

Arthur Kim, Hansol Ko, Wilson Sun
Department of Electrical and Computer Engineering
University of Arizona
Email: akim4@arizona.edu, kohanasol@arizona.edu, wilsonsun@arizona.edu

***Abstract*—Modern key-value (KV) caches struggle with ever-growing working sets: once data exceeds DRAM capacity, performance collapses as requests spill to SSDs. Simply adding more DRAM is expensive in cost and power and eventually hits practical limits. We explore "ZipCache-style" transparent compression on RISC-V (Rocket Chip, BOOM, and VexRiscv) to effectively enlarge the cache without redesigning the entire memory hierarchy. In our Chipyard-based simulations, we model compressed 4 KB pages and a small decompression delay, implemented either as a RoCC-style path or as software NOP stalls. Compared to a baseline DRAM-only cache, our compressed cache roughly halves miss rates under heavy capacity pressure and about doubles modeled throughput on thrashing workloads. A modest decompression cost (about 15 extra cycles per hit) is more than repaid by avoiding slow SSD-backed misses. We also describe practical challenges (broken accelerator paths on BOOM/CVA6, VexRiscv fallback, tuning synthetic workloads so they neither trivially fit nor completely thrash) and outline next steps toward a more realistic design.**

I. Introduction

Data-center KV caches such as memcached or Redis [1], [2] face constant pressure from growing datasets. DRAM capacity does not scale cheaply: more DIMMs mean higher cost, power, and physical footprint. Once the hot working set spills over to the SSDs, tail latency can jump by orders of magnitude because SSD accesses are far slower than DRAM.

This leads to the idea of compression. If many objects contain redundancy (zeros, repeated fields, similar keys), then storing them in compressed form can effectively multiply DRAM capacity without physically adding memory. However, naïvely compressing individual KV items does not work well. Items are often small; compression ratios are modest, and common hash-based indexing scatters objects randomly, destroying spatial locality and making block compression difficult.

Recent work such as ZipCache: A DRAM/SSD Cache with Built-in Transparent Compression [3] tackles this problem for large KV stores by combining DRAM and SSD tiers with transparent hardware compression in the SSD. ZipCache uses a B^+-tree index to group nearby keys, improve compressibility, and offload heavy decompression work to "computational SSDs" instead of the host CPU. The main lesson is that a small amount of extra latency on the cache tier can be acceptable if it avoids much slower SSD misses.

Our project is a small, RISC-V-oriented proof-of-concept inspired by ZipCache. Instead of building a full KV service, we embed a cache simulator in C code running on RISC-V cores inside Chipyard. We compare two designs:

- **Baseline:** DRAM-only cache over 4 KB pages, no compression.
- **"ZipCache" variant:** same DRAM size, but each 4 KB page is assumed to compress by $2\times$, so a 256-page physical cache can hold 512 "logical" pages. Each hit pays an extra decompression delay.

We then construct synthetic workloads that deliberately thrash the baseline cache but largely fit in the compressed cache. The central question is:

When the working set is just beyond DRAM capacity, how much can transparent compression reduce miss rates and improve throughput, and what is the latency cost we pay on hits?

Because this is an ECE 562 final project rather than a production system, our primary goals are to (1) exercise the RISC-V/Chipyard toolchain, (2) build a clean, analyzable experiment, and (3) understand where compression helps or hurts.

II. Related Work

A. On-chip cache compression

Classic CPU-cache techniques like Base-Delta-Immediate (BDI) and Frequent Pattern Compression (FPC) compress cache lines by exploiting simple patterns [4], [5]. BDI stores a base value and small deltas; FPC recognizes common bit patterns like repeated zeros. These schemes target L1/L2 caches with fixed-size lines and traditional set-associative structures. They focus on on-chip hit latency and do not directly address very large KV stores spanning DRAM and SSDs. FPC-style schemes also bring relatively high decompression complexity, which can be problematic when scaled to large memory tiers.

B. KV caching systems

At data-center scale, systems like Meta's CacheLib [6] manage KV caches across DRAM and SSD. CacheLib typically uses hash indexing, which scatters objects across the cache and tends to destroy locality, making it harder to compress blocks effectively. It also suffers from SSD write amplification, since fine-grained updates and hash movements cause extra writes. Kangaroo [7], a CacheLib-based design, introduces

write-ahead logging and other techniques to reduce SSD wear, but it still inherits the hash-indexing structure and its overhead.

C. Memory compression and tiering

Another line of work treats compression as part of the main-memory hierarchy. Transparent Dual Memory Compression (DMC) [8], Linearly Compressed Pages (LCP) [5], and related schemes use fast, low-compression algorithms for hot data and slower, high-compression algorithms for cold data. This creates a tiered approach to compression itself: latency-optimized vs. capacity-optimized tiers. The same philosophy appears in ZipCache, but applied across DRAM and SSD tiers instead of within DRAM alone.

D. ZipCache

ZipCache [3] combines several of these ideas. It uses a B^+-tree index over DRAM and SSD, grouping nearby keys so that objects with similar keys are stored together and compress better. It deploys a "computational SSD" that performs compression and decompression transparently, leaving very hot data uncompressed while heavily compressing colder data. The paper reports substantial throughput improvements and latency reductions by lowering the miss rate into SSD.

Our project borrows ZipCache's central idea—trade a modest decompression delay to avoid expensive SSD misses—but simplifies almost everything else. We do not implement a real compressor, B^+-tree, or SSD device; instead, we emulate a $2\times$ capacity gain in a DRAM-only cache and model decompression as an extra 15 cycles on each hit. The goal is not to compete with ZipCache, but to understand, at an architectural level, when a "compressed cache" is better than a conventional one on RISC-V.

III. Methodology

We build our experiments on top of Chipyard [9], using three RISC-V cores:

- **Rocket Chip:** an in-order core with a simple microarchitecture.
- **BOOM:** an out-of-order core with a deeper pipeline.
- **VexRiscv path:** introduced late in the project as a fallback after we ran into integration issues with CVA6's accelerator interface [10].

The cache behavior itself lives in a C program running on these cores. The program maintains two parallel models:

- A baseline DRAM-only cache that stores 4 KB pages with no compression.
- A ZipCache-style compressed cache with the same physical DRAM size, but allowed to store twice as many "logical" pages.

Both caches use a set-associative organization with LRU replacement. We model a simple two-tier hierarchy:

- **DRAM tier:** fast but limited capacity.
- **SSD tier:** much larger but with about $10\times$ latency compared to DRAM hits.

A. Working-set window and access pattern

To keep the experiment analyzable while still stressing capacity, we generate loads to pages drawn from a configurable working-set window. For each run, we issue 2000-page accesses. Our metrics are:

- Average latency per operation (modeled cycles),
- Throughput (operations per million core cycles),
- DRAM and SSD miss rates.

A key step is choosing the working-set window so that the two caches sit in different regimes:

- **Small window ($\approx$200 pages):** fits in both the 256-page baseline cache and the 512-page effective ZipCache. Neither design is under real pressure.
- **Ideal window ($\approx$350 pages):** small enough to fit in the 512-page ZipCache, but large enough to cause noticeable thrashing in the 256-page baseline.
- **Large window ($\approx$600 pages):** significantly larger than 256 pages, so the baseline is well past its capacity and thrashes heavily, while the 512-page ZipCache can still hold most of the working set.

We use a pseudo-random stride within the window so that we do not accidentally help the baseline cache with simple sequential locality. The idea is that whether a request hits or misses should be dominated by capacity, not by a particularly friendly access pattern.

B. Latency model and decompression overhead

To keep the math simple but realistic in order of magnitude, we assume:

- DRAM hit latency: 50 cycles,
- SSD access (on a miss): 500 cycles.

In the baseline cache, a hit pays 50 cycles; a miss pays 500 cycles.

In the ZipCache, a hit pays 50 cycles plus a decompression penalty of about 15 cycles, for a total of 65 cycles. We model this extra cost differently depending on the core:

- On Rocket, we approximate decompression via a small RoCC-style instruction sequence that burns cycles.
- On BOOM and VexRiscv, where the accelerator template path was not clean, we insert NOPs to stall the core for the same number of cycles.

This gives us a clean comparison: a "faster but smaller" baseline cache versus a "slower but bigger" ZipCache.

IV. Experimental Setup

For each core (Rocket, BOOM, VexRiscv) we run the same C workload under:

- Two cache configurations: Baseline vs. ZipCache,
- Three window sizes: small (200 pages), ideal ($\approx$350 pages), and large (600 pages).

Between runs we flush the cache model so that results are not polluted by previous state. For each run we record:

- Total modeled cycles,
- Average latency per request,

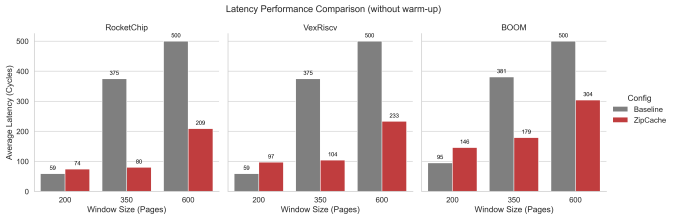

Fig. 1. Latency performance comparison without warm-up.

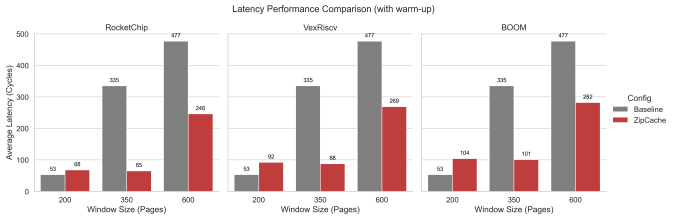

Fig. 2. Latency performance comparison with warm-up.

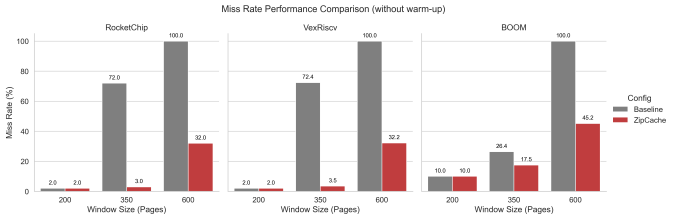

Fig. 3. Miss-rate performance comparison without warm-up.

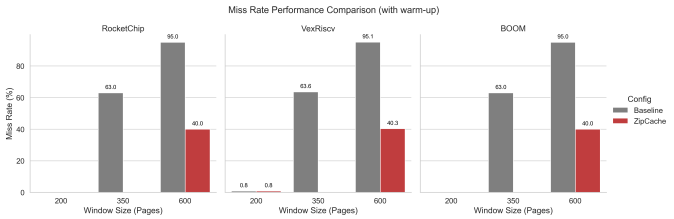

Fig. 4. Miss-rate performance comparison with warm-up.

- Throughput (ops per million modeled cycles),
- DRAM hits/misses and SSD accesses for each cache.

Although the cores differ in microarchitecture, the cache model and access sequence are identical, so we can focus primarily on the relative behavior of Baseline vs. ZipCache. The Rocket results are easiest to interpret, and BOOM/VexRiscv show similar qualitative trends.

The report includes a table summarizing the key metrics (miss rates and throughput) for small and large windows. The ideal window case falls between these extremes and highlights the sweet spot where compression provides the largest benefit.

V. Results

We evaluate our ZipCache-inspired compressed cache on three RISC-V cores (RocketChip, BOOM, and VexRiscv) under three working-set sizes (200, 350, and 600 pages. For each configuration we measure average latency per request, DRAM miss rate, and throughput (operations per million modeled cycles). Figures 1–6 summarize the results. Figures 1, 3, and 5 show cold-start behavior without any cache warm-up, while Figures 2, 4, and 6 report steady-state behavior after a warm-up phase that touches twice the working-set window.

Overall, the figures tell a consistent story: when the working set already fits in DRAM, compression only adds decompression overhead, but once the working set exceeds the baseline DRAM capacity, the extra effective capacity of ZipCache roughly halves miss rates and substantially improves throughput across all three cores.

A. Latency

Figures 1 and 2 compare the average modeled latency per request for the baseline and ZipCache configurations. With warm-up and a small 200-page window (Figure 2), both caches achieve essentially 0% miss rate on all cores, so ZipCache's decompression delay is pure overhead. On RocketChip, for example, average latency increases from about 53 cycles in the baseline to 68 cycles with ZipCache; BOOM and VexRiscv show similar behavior. In this "fits-in-DRAM" regime, compression should be disabled.

As the window grows to 350 pages, the baseline cache begins to thrash while ZipCache still fits the working set. Figure 2 shows that average latency on RocketChip jumps to roughly 335 cycles in the baseline, but only around 65 cycles with ZipCache. BOOM and VexRiscv exhibit the same pattern: baseline latency in the few-hundred-cycle range versus roughly 100 cycles with ZipCache. For the 600-page window, both caches are under heavy capacity pressure, yet ZipCache still cuts average latency by roughly 2× (e.g., from about 477 cycles to about 250–280 cycles depending on core). The no-warm-up curves in Figure 1 follow the same qualitative trends, but with slightly higher latencies in the small-window case due to cold misses during cache fill.

B. Miss rate

Figures 3 and 4 plot DRAM miss rate as a function of window size. After warm-up (Figure 4), the 200-page window yields near-zero miss rate for both configurations and all cores, confirming that the working set fully fits in DRAM. The difference between baseline and ZipCache is therefore entirely due to decompression overhead, not hit/miss behavior.

Once the window reaches 350 pages, the baseline miss rate rises to roughly 60–65% on all cores, while ZipCache keeps the miss rate near 0%. This is the "sweet spot" where ZipCache's doubled effective capacity prevents most SSD accesses. At 600 pages, the baseline miss rate climbs to about 95%, but ZipCache still limits misses to roughly 40–45%. The no-warm-up miss-rate curves in Figure 3 again show higher apparent miss rates for the small window (because the cache is still warming), but converge to the same three regimes—fits, sweet spot, and thrashing—after enough requests.

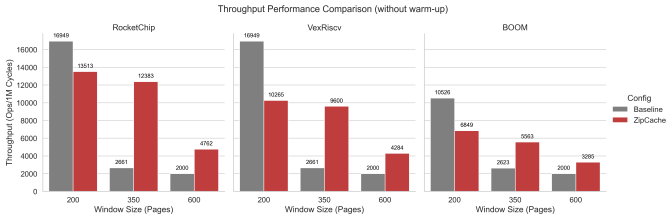

Fig. 5. Throughput performance comparison without warm-up.

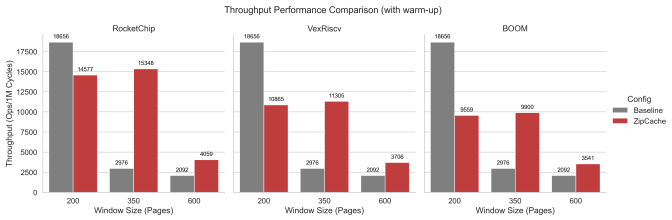

Fig. 6. Throughput performance comparison with warm-up.

C. Throughput

Figures 5 and 6 report throughput in operations per million modeled cycles. With warm-up and a 200-page window (Figure 6), baseline throughput is highest: on RocketChip it reaches about 18.6k ops/Mcycle, while ZipCache achieves around 14.6k ops/Mcycle due to the extra decompression delay. BOOM and VexRiscv show similar reductions in throughput when compression is enabled in this capacity-plenty regime.

In the 350-page window, throughput trends reverse. Because ZipCache converts almost all SSD misses into DRAM hits, it delivers a 3–5× throughput improvement over the baseline on all cores. For example, on RocketChip baseline throughput is only about 3.0k ops/Mcycle, whereas ZipCache reaches roughly 15.3k ops/Mcycle. BOOM and VexRiscv show qualitatively similar gains. In the 600-page window, ZipCache nearly doubles throughput compared to the heavily thrashing baseline (e.g., from about 2.1k ops/Mcycle to about 3.5–4.1k ops/Mcycle, depending on core). The no-warm-up throughput curves in Figure 5 again suffer in the small-window case because the caches are still filling, but align with the warm-up results once steady state is reached.

D. Cross-core summary

Across RocketChip, BOOM, and VexRiscv (used as a stand-in for the originally planned CVA6 path), the six figures highlight the same trade-off:

- When the working set fits within DRAM, ZipCache's decompression penalty simply slows the system down.
- When the working set is slightly larger than DRAM capacity, ZipCache's extra effective capacity nearly eliminates SSD misses, reducing average latency by several× and boosting throughput by similar factors.
- When the working set is much larger than DRAM, both caches thrash, but ZipCache still recovers a significant fraction of performance by cutting miss rate roughly in half.

In other words, compressed caches are most beneficial in the capacity-limited regime where DRAM is tight but not hopelessly overrun. Our results on three different RISC-V cores suggest that this qualitative behavior is robust to microarchitectural details.

VI. Limitations

Our prototype has several important limitations and should be viewed as a proof-of-concept rather than a full system.

A. No real compressor or decompressor

We model a fixed 2× capacity gain and a fixed 15-cycle decompression overhead; neither is measured from a real algorithm. We ignore issues like bandwidth consumption and energy cost of a real compression engine.

B. Metadata and tag-store limits

In reality, increasing effective capacity also requires more tags or metadata entries. Prior work such as the BDI paper shows that doubling the number of tags can significantly increase tag-store size. Our 512-page cap for ZipCache is a simple way to model this metadata cost: even if data compresses well, the cache cannot track more than 512 logical pages without more index storage.

C. Simplified indexing and layout

We do not implement ZipCache's B^+-tree index or its grouping of nearby keys. Instead, we treat each 4 KB page independently and assume that compression works equally well for all pages. We do not model write amplification, complex eviction policies, or interactions with SSD firmware.

D. Synthetic, read-only workloads

Our workloads are intentionally synthetic and adversarial: pseudo-random access within a fixed window, read-only traffic, and a clean separation between DRAM hits and SSD misses. Real KV caches see skewed key popularity, mixes of reads and writes, and background maintenance tasks. These factors could change the trade-offs.

E. Core and accelerator integration gaps

Our original plan was to integrate a RoCC-like decompression accelerator on Rocket, BOOM, and CVA6. In practice, the accelerator path worked well only on Rocket. BOOM required more plumbing than we could finish in a semester, and CVA6's interface caused persistent bring-up issues. As a workaround, we modeled decompression delay with NOPs and shifted part of the evaluation to a VexRiscv core, which is not identical to CVA6. This means the interaction between core, cache, and memory system is simplified.

Despite these limitations, the prototype still gives us useful architectural insight into when cache compression is worthwhile.

VII. Conclusions and Future Work

Despite its simplifications, our project supports a clear conclusion: on RISC-V, adding transparent compression to a cache can dramatically reduce miss rates and recover throughput when the working set slightly exceeds DRAM capacity. In our modeled runs, a small decompression delay (about 15 cycles) was more than repaid by the hundreds of cycles saved by avoiding SSD misses. When the data already fits in DRAM, however, compression only adds overhead. This suggests that compressed caches are most attractive in memory-constrained scenarios where capacity pressure is real.

For future work, we see several directions:

- **Replace NOP-based delay with real hardware.** Integrate an actual decompression engine, either as a RoCC accelerator or a TileLink-attached peripheral, so that we can measure real latency, bandwidth, and contention effects.
- **Use more realistic KV workloads.** Move beyond synthetic windows to traces that include skewed key distributions, mixed reads and writes, and background operations. This would test whether the ZipCache-style benefits survive in messy, real-world conditions.
- **Refine compression policies.** Add simple policy improvements such as detecting all-zero pages (encode with a single bit and skip decompression entirely) or dynamically switching between "fast/low-compression" and "slow/high-compression" algorithms, inspired by DMC.
- **Study indexing and write amplification.** Implement a lightweight B^+-tree or similar structure to group keys and evaluate how it affects compressibility, eviction, and write-back traffic to SSD.

Overall, our "Cache(d) Compression" project suggests that modest hardware and software changes could let future RISC-V systems stretch limited DRAM much further, as long as the workload sits in the sweet spot where DRAM is tight but not hopelessly overrun.

References

[1] "memcached: A distributed memory object caching system," [Online]. Available: https://memcached.org/. Accessed: Dec. 7, 2025.

[2] "Redis: In-memory data structure store," [Online]. Available: https://redis.io/. Accessed: Dec. 7, 2025.

[3] R. Xie, L. Ma, A. Zhong, F. Chen, and T. Zhang, "Zip-Cache: A DRAM/SSD cache with built-in transparent compression," in *Proc. Int. Symp. on Memory Systems (MEMSYS)*, 2024.

[4] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. 21st Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 377–388.

[5] G. Pekhimenko, V. Seshadri, O. Mutlu, M. A. Kozuch, P. B. Gibbons, and T. C. Mowry, "Linearly compressed pages: A low-complexity, low-latency main memory compression framework," in *Proc. IEEE/ACM Int. Symp. on Microarchitecture (MICRO)*, 2013.

[6] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger, "The CacheLib caching engine: Design and experiences at scale," in *Proc. 14th USENIX Symp. on Operating Systems Design and Implementation (OSDI)*, 2020.

[7] S. McAllister, B. Berg, D. S. Berger, N. Beckmann, M. Harchol-Balter, and G. R. Ganger, "Kangaroo: Theory and practice of caching billions of tiny objects on flash," *ACM Trans. Storage*, vol. 18, no. 3, Art. 21, pp. 1–33, Aug. 2022.

[8] S. Kim, S. Lee, T. Kim, and J. Huh, "Transparent dual memory compression architecture," in *Proc. 26th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 206–218.

[9] U. C. Berkeley Architecture Research Group, "Chipyard: An integrated SoC design framework for agile hardware development," [Online]. Available: https://chipyard.readthedocs.io/. Accessed: Dec. 7, 2025.

[10] C. Papon, "VexRiscv: A generated RISC-V CPU implemented in SpinalHDL," [Online]. Available: https://github.com/SpinalHDL/VexRiscv. Accessed: Dec. 7, 2025.