Dynamic Cache Compression

Hassan Farooq, Spenser Fong, Timothy Vitkin

Background

- Many workloads benefit from aggressive prefetching techniques, but these techniques pollute the cache
- Cache capacity is a limiting factor
- Cache compression has been around for a while but decompression has a very high overhead
- We want to explore ways to dynamically hide cache decompression latency

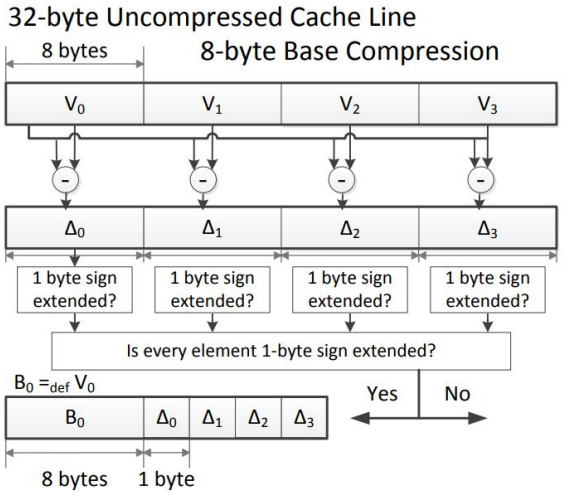
Previous Work

- Most cache compression done in LLC - increase effective cache capacity, keeps latency/physical cache size low
- Some work on partially compressed caches
- Work on compression algorithms for different data types, reduce energy consumption
- Reduce latency seen with decompression
    - Compression in addition to pre-fetching
    - Selective compression, parallel decompression, decompression buffers

Compression

- Base-Delta Immediate (BDI) compression
  - High compression ratio
  - Low decompression latency
  - Modest hardware overhead
- Most papers use this algorithm, and it is available in Gem5

Cores	Avg Speedup Over No Compression[1]
1	5.1%
2	9.5%
4	11.2%



BDI design[1]

[1] 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 2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT), 2012, pp. 377–388.

Original Design

- Use Runahead execution in Gem5 to hide cache decompression latency
- Overview of Gem5 Fetch stage:
    1. Instruction enters Fetch stage
    2. CPU checks if there is an available thread to fetch
        a. Ie, if a thread is currently idle / blocked, uses that thread to fetch
        b. If there is no available thread, stall
    3. CPU Sends a timing request to cache with information about what to fetch

Original Design continued

1. Cache receives timing request:
    a. If cache hit, send back data + simulated cache hit time
    b. if cache miss, and not LLC, send timing request to next-level cache
    c. If cache miss, and LLC send timing request to TLB (where mem access occurs). Checkpoint this thread's state, and execute waiting instructions in the I-Queue until received response from miss.
    d. Once received data response, restore thread state and resume execution with the fetched data

Challenges

- For runahead, we need to checkpoint the system state…
- …but gem5 does not allow writes to a thread state without flushing the entire pipeline!
- Were unable to implement runahead because we could not figure out how to restore system checkpoints

Idea #2

Use Gem5’s hardware threading for SMT

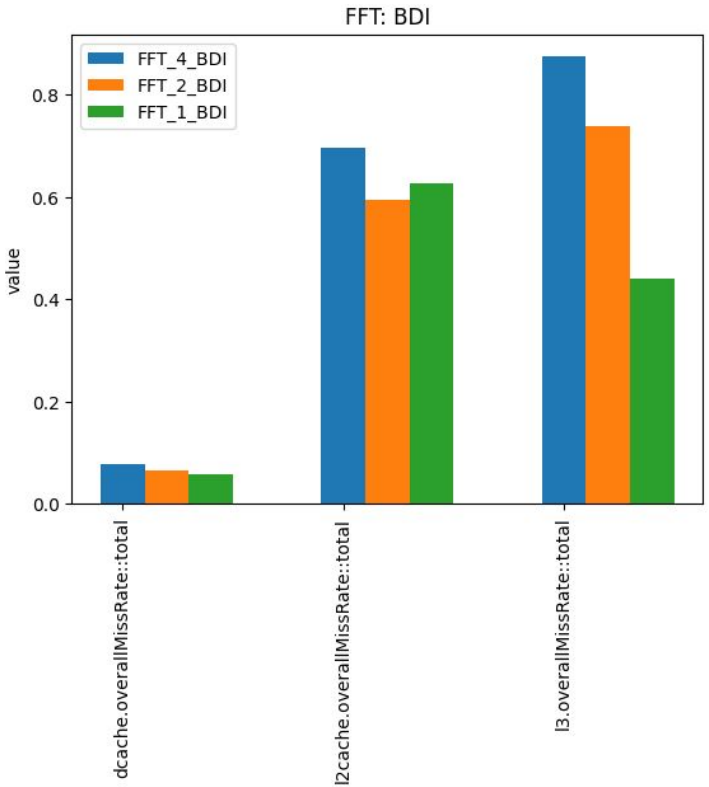
- Round-Robin SMT: If a thread is not doing work, make it fetch!
- Compare SMT performance on different workloads with cache compression and without cache compression

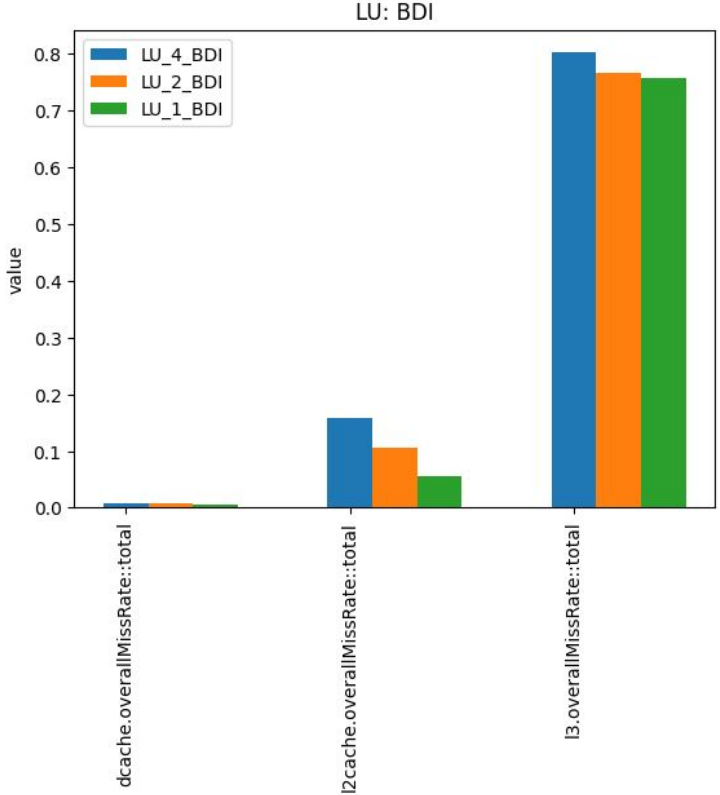
Results

- Core count (1 vs 2 vs 4 core) and effect on compression performance
- Compressed L3 alters performance in other caches
- Number of prefetching threads and effect on performance
- Find workload that isn’t benefitted from given compression algorithms
- X86 vs RISC-V compressed cache effects

Metric (2-core system, 8 prefetching threads)	**Compressed**	**Uncompressed**	**% Difference**
system.cpu0.icache.overallHits::total	7.24038e+06	3.65178e+06	65.89
system.cpu1.icache.overallHits::total	3.49762e+06	1.76983e+06	65.6

Measured Workload Comparison





Future Work

- How does varying levels of cache hierarchy affect compression performance?
    - How does mixing/matching different compression algorithms on different levels affect performance?
- How does compression fare in multi-core processors vs. single-core?
- Developing better support for making compression algorithms in simulators
- (Given a workload that isn't benefiting from the given compression algorithms)

  How to develop a compression algorithm to address that workload?