Effectiveness of Cache Compression in Multi-Core RISC and CISC Systems

Hassan Farooq
University of Illinois
hfaroo9@illinois.edu

Spenser Fong
University of Illinois
sfong5@illinois.edu

Timothy Vitkin
University of Illinois
tvitkin2@illinois.edu

***Abstract*—We present a discussion on the effectiveness of various cache compression algorithms and CPU configurations on CPU performance. We determine that for a large compressed L3 cache, cache compression has a minimal effect on the performance of the CPU. We show performance comparisons between different workloads and their effect on miss rate of L3 cache. We also show that higher CPU core counts result in a higher Last-Level Cache (LLC) miss rate due to the aggressiveness of the prefetcher used. Simulations on both RISC (RISC-V) and CISC (x86) architectures were performed. We discuss a failed approach to use runahead execution in order to dynamically partition data in the cache prior to its use.**

I. Introduction

One way to exploit the benefits of larger caches without increasing the area the cache takes up is cache compression, which compacts data stored in the cache. Cache compression can lead to higher data bandwidth and reduced overall energy consumption, at the cost of hit latency. Reducing overhead between memory requests for compressed data in caches and the data being returned to the main core can be beneficial to performance and allow for similar compression techniques to be applied in additional levels of cache. This could further increase effective cache capacity.

There is no overview of cache compression and its effects on different architectures. Pekhimenko et al. [1] discusses the performance of BDI compression on various x86 core configurations, but there is no such overview for either other cache compression algorithms nor for other CPU configurations.

Throughout this paper, we explore various cache compression algorithms and their resultant performance in various configurations of RISC (RISC-V) and CISC (x86) CPUs. We use the gem5 simulator [2] to run test programs and measure the cache performance in terms of cache miss ratios and CPU performance in terms of instructions per cycle (IPC). Our testing methodology is discussed in Section IV. Our results are shown in Section VI.

II. Background and Related Work

Accessing data in main memory can be very costly as it can take a significant amount of clock cycles, especially in comparison to data access from caches. Increasing cache size is not always a viable alternative in order to move more data closer to the main core simply due to limited die area, additional cost, additional heat, and additional power necessary. Among various methods, cache compression has been a popular technique to artificially increase the amount of cache available without substantially increasing the physical size of the caches. Even with the additional computation overhead of compression and decompression, cache compression can be significantly faster than main memory access and thus beneficial for performance. Due to the overhead involved, cache compression techniques tend to mainly be utilized in last-level cache (LLC) [3]. The issue here is that while less space is necessary for the same amount of data, more time is needed before the data is in a state in which it can be used (ie. uncompressed data) in comparison to a traditional cache. Mitigating this overhead could allow for decreased wait time for memory and improved performance and provide justification in using similar techniques in lower level caches, thus further artificially increasing the amount of space available in the caches.

Compressing data in caches close to the main core is tricky due to the overhead and additional latency that occurs with the decompression necessary before data can be used [4]. Due to this, most recent work investigates compression techniques in the LLC of the processor in order to increase effective cache capacity while keeping latency and physical cache size low [3]. This is especially important when using an aggressive, stride-based prefetcher that pollutes the cache, causing degradation in performance. In such cases, compression can provide performance improvements of up to 50% [5].

Previous work on using cache compression have also investigated partially compressed caches in which the cache has a partition for compressed data and a separate partition for uncompressed data [6], [7]. Some compression algorithms have been explored to increase compression speed, account for different data types, and reduce energy consumption [8]–[10]. For instance, by using chunks of four cache lines ("superblocks"), Sardashti et al. were able to reduce tag overhead in compressed caches, resulting in improved effective cache capacity without the need of significant metadata, backward pointers, or the complexity of skewed associativity [11].

Compressed LLCs that use data criticality as a consideration during compression have also been proposed, leading to 4MB compressed LLCs having performance comparable to that of an 8MB uncompressed LLC [12]. Combining compression with various replacement policies has also been explored to try and improve performance. It was found that advanced replacement policies interact poorly with compression, resulting in

no noticeable improvements between compressed and uncompressed caches, but with an opportunistic cache compression mechanism, improvements of up to approximately 9% could be seen [13]. Techniques other than compression have also been tested to try and increase effective cache capacity. For instance, Huang et al. proposed a critical-words-only cache in which only words that are generally accessed before others are kept in cache resulting in a 256kB L2 cache performing just as well as a 512kB conventional L2 cache on average [14].

Due to the delay of decompression, certain techniques have been explored in order to hide the latency penalty incurred during this step [15]. Alameldeen et. al proposed a technique where an LRU and compressed data size determined whether or not data should be stored compressed or decompressed, providing an improvement of 17% for memory-intensive benchmarks [15]. They also note that while on a typical compressed cache, a memory-intensive benchmark with a low cache miss rate incurred a performance penalty of 18%, their adaptive-compression cache only caused a 0.4% performance penalty [15]. Rea et al. proposed cache compression in addition to data prefetching in order to offset some of the latency caused by decompression specifically in L1 caches [16]. It was found that a Base-Delta-Immediate (BDI) compression combined with stride/last outcome prefetching allowed for a 1.7% average speedup compared to simply using compression without prefetching [16]. Lee et al. suggested using selective compression, parallel decompression, and the use of decompression buffers to reduce decompression overhead resulting in a 35-53% reduction in data traffic and a 20% reduction in average memory access time [17].

While there are many different cache compression algorithms already implemented within the gem5 simulator, BDI compression is generally regarded as the best overall [1]. This is due to its high compression ratio, low decompression latency, and modest hardware overhead. At a high level, the stored values in the cache line have small differences between them for most cache lines, and a base value and an array of differences whose combined size is much smaller than the original cache line can be used to represent that original cache line [1].

III. Design

The original proposed design involved using runahead execution in gem5 to decrease the inherent latency that comes with cache decompression. Instructions would enter the fetch stage. The CPU checks if there is an available thread to fetch: if a thread is currently idle or blocked, the CPU will use that thread for fetch, and if there is no available thread, the CPU will stall. After finishing this condition, the CPU would send a timing request to the cache hierarchy with information about which data to fetch. Once the cache receives the timing request, there are three events that could happen:

- If there is a cache hit, send data back and record the simulated cache hit latency
- If there is a cache miss and not LLC, send a timing request to the next-level cache
- If there is a cache miss and the LLC sent a timing request to the TLB when a memory access occurs, checkpoint the thread's state, and execute pending instructions in the instruction queue until a response from the miss has been received

Once a data response has been received, the thread state will be restored and execution will resume with the fetched data.

There were some challenges that were encountered while implementing the original design, which are discussed in Section IV. Due to this, the project pivoted to another design. The new design uses the hardware threading provided in gem5 for simultaneous multithreading (SMT). If there are idle threads, then one of those threads determined by a round robin SMT scheme will fetch new instructions. Performance from SMT would be compared between different workloads with and without cache compression. The BDI compression algorithm was chosen to be the main algorithm to test against the baseline. Testing methodology is explained in Section V.

IV. Implementation/Methodology

The original project required the ability to checkpoint the processor system state for runahead execution. However, a problem presented itself in gem5. The simulator does not allow writes to a thread state without flushing the entire pipeline, which defeats the purpose of using runahead execution. As a result, runahead execution was not able to be implemented, and no valid method for restoring system checkpoints in gem5 was discovered. Rather, it was decided to investigate the effect of core count and architecture type on cache compression. This was done by adding support for compression in L3 cache in the gem5 simulator then running numerous simulations and analyzing the results.

In order to add L3 cache with compression support, some of the configuration Python files within the gem5 simulator needed to be changed. Specifically, gem5/configs/common/Caches.py was edited to add an L3 cache with default options identical to that of the default L2 cache except with associativity of 16, mshrs set to 512, tgts_per_mshr set to 20, write_buffers set to 256, tags set to CompressedTags(), and compressor set to BDI(). Additionally, gem5/configs/common/CacheConfig.py was edited to add an L3 cache to the processor being simulated and to be able to parse compression attributes set by the user. gem5/configs/common/Options.py was also changed to support passing in L3 cache configuration options when starting the simulation through the command line (size, associativity, tag type, and compression algorithm). Finally, gem5/src/cpu/BaseCPU.py was edited to have 2 hardware thread contexts rather than the default of 1. An additional Python script was also written in order to automate the process to run all necessary simulations.

Simulations were run on three different test programs (FFT, LU, RADIX), on three different core counts (1, 2, 4), for both x86 and RISC-V architectures, and for all compression algorithms included by default in gem5. These compression algorithms included BaseCacheCompressor,

BaseDictionaryCompressor, Base64Delta8, Base64Delta16, Base64Delta32, Base32Delta8, Base32Delta16, Base16Delta8, CPack, BDI, Frequent Pattern Compression (FPC), Frequent Pattern Compression with limited Dictionary support (FPCD), FrequentValuesCompressor, MultiCompressor, PerfectCompressor, RepeatedQwordsCompressor, and ZeroCompressor. A simulation run was also done for all combinations of the test programs, core counts, and architectures with no cache compression to be used as a baseline. This resulted in a total of 324 simulation runs.

For all simulations, the processor was set with a 3GHz CPU clock, a 64 byte cacheline size, and 3 levels of cache. L1 instruction and data caches were set to a size of 32kB with an associativity of 8. L2 cache was set to a size of 2MB and an associativity of 16. L3 cache was set to a size of 6MB and associativity of 24, with the compression algorithm used changing based on the current simulation run. Only L3 cache was compressed. For simulations with more than one core, L1 and L2 caches were per-core caches, while L3 was shared across all cores. With x86 simulations, the default gem5 X86O3CPU CPU type was used while with the RISC-V simulations the default gem5 RiscvO3CPU CPU type was used. When running the FFT program, the -m16 flag was passed in, with the LU program the -n512 flag was passed in, and with the RADIX program the -n1048576 was passed in.

Core counts and architecture were chosen as variables to change as many previous papers did not discuss how altering core counts nor how RISC versus CISC architectures affected the effectiveness of cache compression. On the other hand, many papers discussed cache size and cache levels and their effects on cache compression, something the simulations run for this paper did not address. Future work could investigate additional cache sizes, compression on cache levels other than the LLC, and varying CPU clock speeds.

All source code can be found in this GitHub repository: https://github.com/s-hfarooq/ECE511-Dynamic-Cache-Decompression/. Simulation outputs can be found within the gem5/benchmarks/results/ folder of the same repository.

V. Evaluation

During analysis, the BDI compression algorithm was used during comparisons as this is considered one of the better cache compression algorithms provided in the simulator.

Python scripts were written in order to easily compare various simulation statistics and how various processor and cache characteristics altered performance when caches were compressed. IPC, memory bandwidth, and cache hit/miss rate were the most significant statistics that were examined in the simulation outputs. One such statistic that was investigated was the effects of core count on cache compression for each compression algorithm listed in Section IV. Figure 1 and 3 show the output of the RISC-V FFT run using BDI compression in L3 cache. Figure 2 and 4 show the output of the RISC-V FFT run with no cache compression. Figure 5 and 7 show the output of the x86 FFT run using BDI compression in L3 cache. Figure 6 and 8 show the output of the x86 FFT run with no cache compression. A subset of simulation results from some of the compression algorithm runs for the FFT test program can be found in Table I. Full simulation results can be found in the GitHub repository linked in Section IV.

Across all tests, including those not shown in the provided graphs, 4 core processors tended to have greater cache miss rates compared to single core processors for all benchmarks run, on both x86 and RISC-V architectures, and across all compression algorithms. This may be due to a data locality issue as L1 and L2 caches were per core and thus some cores may have needed to fetch data computed by another core at some points in the test program. On occasion it was observed that 2 core systems tended to have the lowest cache miss ratio as can be seen in the L2 cache in Figures 1, 2, 5, and 6. Cache miss rate as well as IPC across all workloads tested were fairly similar when comparing the various compression algorithm simulations to the baseline uncompressed cache run. Across all tests, there was less than a 1% increase in L3 cache miss ratio between the compressed and uncompressed cache simulation results. This could be due to the fairly large caches that were used during simulation as well as the specific test programs that were used. It may be beneficial to rerun these simulations using the same benchmarks used in other papers, most notably the SPEC CPU 2017 benchmark suite [18]. This was not used during the simulations run for this paper as they are not freely available.

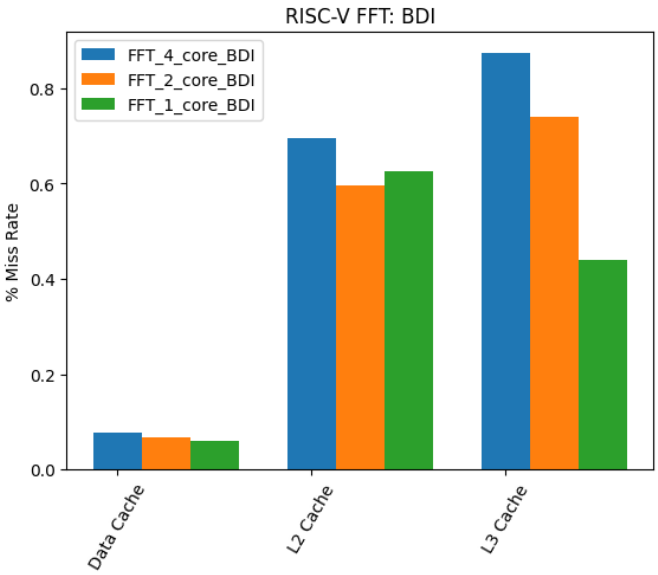

Fig. 1: RISC-V FFT simulation with BDI compression in L3 cache

VI. Discussion

Our results show that when working with compressed L3 cache, there is minimal impact on L3 cache miss rate. When working with the BDI compressor, there is less than a 1%

		Architecture / Core Count					
		RISC-V / 1	RISC-V / 2	RISC-V / 4	x86 / 1	x86 / 2	x86 / 4
No Compression	Data Cache Miss Rate	0.058602	0.0667715	0.07793725	0.029289	0.052064	0.06711075
	L2 Cache Miss Rate	0.626482	0.595182	0.69591625	0.628338	0.6041885	0.7027605
	L3 Cache Miss Rate	0.439792	0.739778	0.873701	0.460246	0.754606	0.862833
	IPC	1.598053	1.823887	1.93100975	1.54521	1.5635815	1.5461945
BDI	Data Cache Miss Rate	0.058591	0.0667715	0.07792675	0.029289	0.0520645	0.06711275
	L2 Cache Miss Rate	0.626484	0.595182	0.69613225	0.628338	0.604177	0.7027245
	L3 Cache Miss Rate	0.439788	0.739778	0.874391	0.460246	0.754585	0.863046
	IPC	1.598053	1.823887	1.92973775	1.54521	1.5636075	1.54618375
FrequentValuesCompressor	Data Cache Miss Rate	0.058588	0.066839	0.0778915	0.02929	0.052065	0.0671115
	L2 Cache Miss Rate	0.626484	0.5951985	0.69602425	0.628324	0.604183	0.7027235
	L3 Cache Miss Rate	0.439788	0.73952	0.874052	0.460254	0.754606	0.863019
	IPC	1.598016	1.8238205	1.9326	1.545175	1.563671	1.54620625
CPack	Data Cache Miss Rate	0.058588	0.0667885	0.0779115	0.02929	0.052065	0.06711225
	L2 Cache Miss Rate	0.626488	0.5952565	0.69609075	0.628251	0.604174	0.702754
	L3 Cache Miss Rate	0.43978	0.739674	0.874658	0.460296	0.754585	0.862846
	IPC	1.598014	1.823648	1.93344825	1.545176	1.563662	1.546105
FPCD	Data Cache Miss Rate	0.058589	0.0670405	0.07779475	0.029289	0.052065	0.067124
	L2 Cache Miss Rate	0.626484	0.5941985	0.696014	0.628286	0.604177	0.70284
	L3 Cache Miss Rate	0.439788	0.741271	0.874582	0.460281	0.754585	0.862251
	IPC	1.598037	1.824564	1.933534	1.545176	1.563602	1.54603375
MultiCompressor	Data Cache Miss Rate	0.058588	0.06684	0.07783525	0.029289	0.052065	0.06710125
	L2 Cache Miss Rate	0.626488	0.739818	0.69600075	0.628329	0.604174	0.70299325
	L3 Cache Miss Rate	0.43978	0.739818	0.874701	0.460258	0.754585	0.861784
	IPC	1.598029	1.823809	1.9332856	1.545176	1.5636755	1.54635325

TABLE I: FFT Simulation Results (Partial) - Full simulation statistics can be found in the GitHub repository

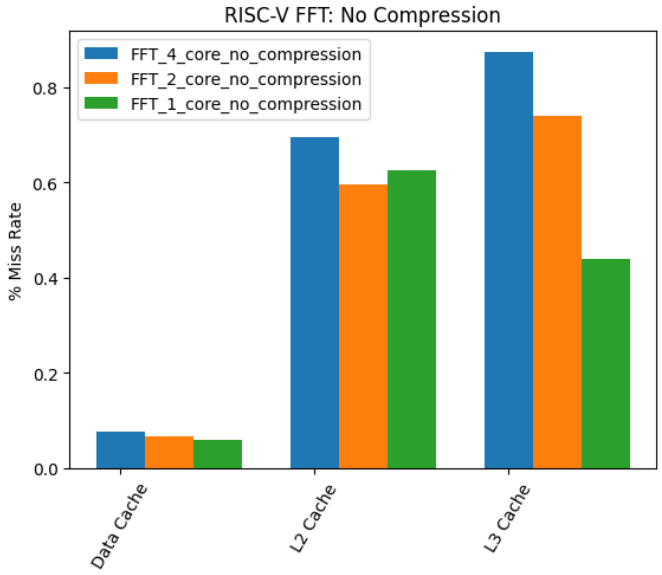

Fig. 2: RISC-V FFT simulation with uncompressed L3 cache

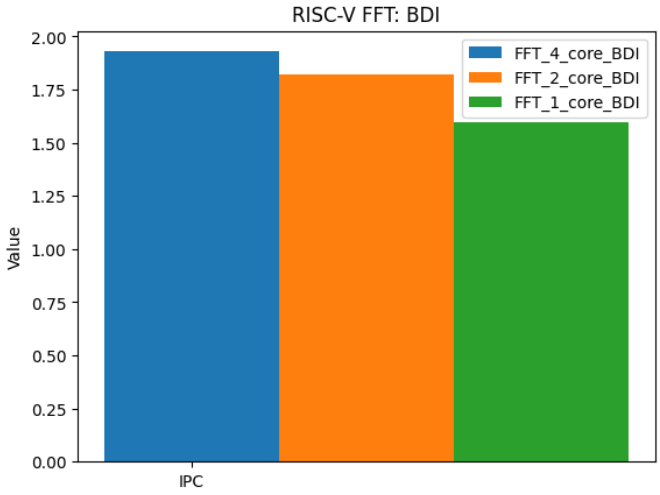

Fig. 3: IPC of RISC-V FFT simulation with BDI compression in L3 cache

difference between BDI compression and uncompressed L3 cache. There is also no major impact on IPC with a cache compression algorithm. However, this is promising: a lack of impact to IPC means the cache decompression latency is very minimal and does not severely impact the operation of the CPU.

Our results also show that more cores generally correlates with a higher miss rate (Figures 9, 10). This is likely due to more cache pollution, since in our testing methodology we used an aggressive prefetcher. Since all cores had the same prefetcher, it is very likely that the L3 cache was polluted with the outputs of the prefetcher.

We learned some valuable lessons during the development of this project. First, we spent a lot of time trying to modify the Fetch stage of the gem5 simulator so that it fit our needs. However, looking back after we pivoted the project, we found a ”SimpleThread” class already built-in to gem5 with sample usages. If we were to go with our initial idea, instead of fighting with the complex gem5 Pipeline, we could instead modify one malleable part of the simulator, making our lives easier and simplifying the process of development. This leads to one of our key takeaways of this project: when modifying an existing simulator - try to make the smallest modifications

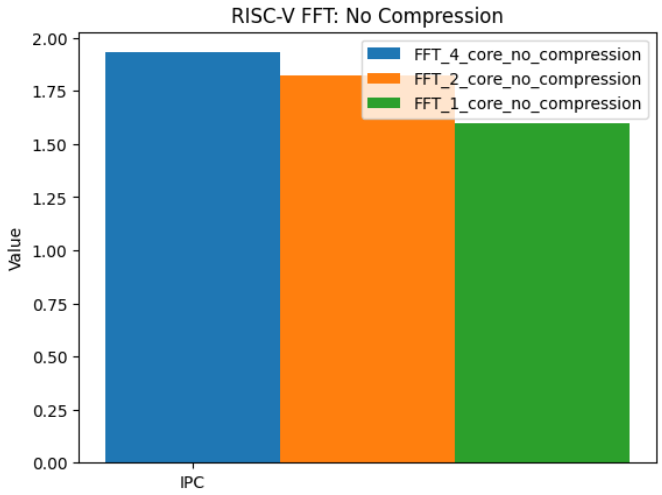

Fig. 4: IPC of RISC-V FFT simulation with uncompressed L3 cache

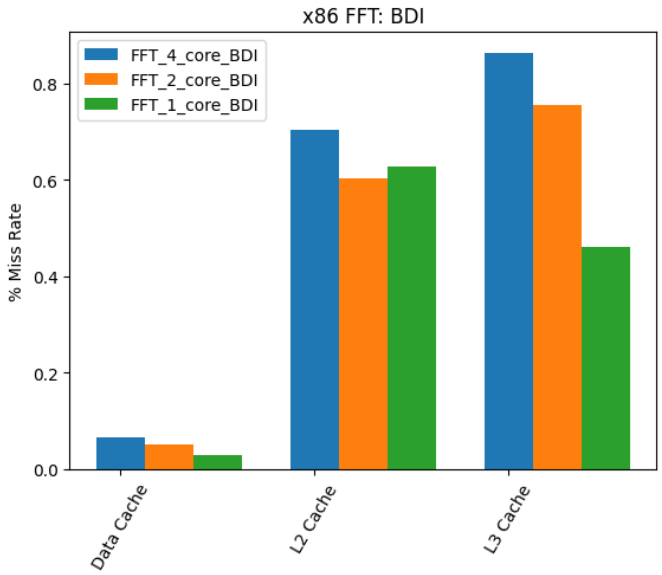

Fig. 5: x86 FFT simulation with BDI compression in L3 cache

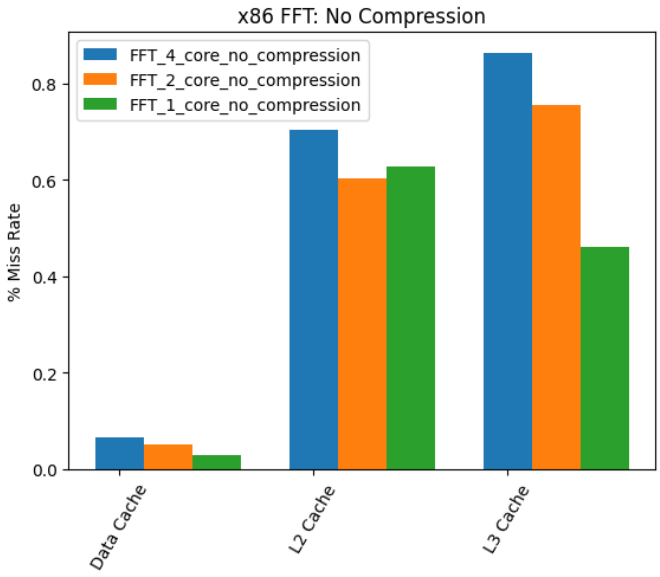

Fig. 6: x86 FFT simulation with uncompressed L3 cache

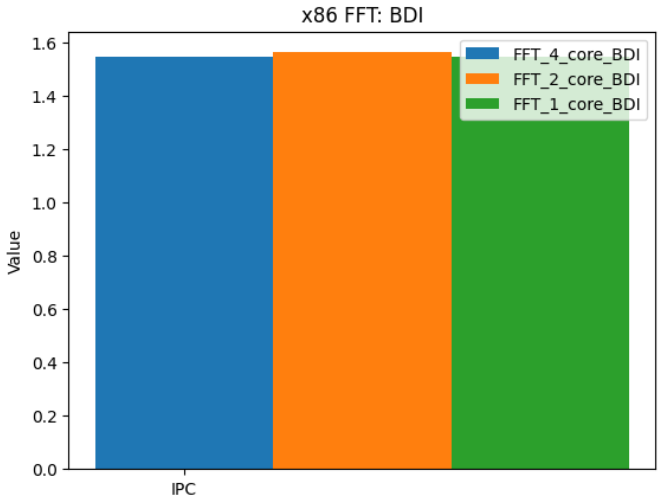

Fig. 7: IPC of x86 FFT simulation with BDI compression in L3 cache

possible.

VII. Conclusion

Looking back at the statistics collected, it is unclear whether or not compression is greatly affected by processor architecture or core count. The results collected showed less than a 1% difference between cache miss rates between compressed and uncompressed caches. In comparison to previous papers which generally saw at least a 5% difference between the two, there is clearly some difference in the way simulations were ran. It is likely this difference lies in the test programs used as well as cache sizes simulated. Due to the delta between results collected through these simulations and those from other papers, we cannot conclude how the viability of cache compression is affected by either processor architecture (RISC vs CISC) nor by core count. In the future, using extremely small caches to truly see the benefits of cache compression across all test programs would be a smart move. Simulating a larger variety of test programs would also be beneficial to truly understand what types of tasks benefit from cache compression. Finally, a look into cache sizes and its impact on cache compression viability would also be interesting to investigate.

VIII. Future Work

Given the nature of the work done in this project, ways to develop better support for making compression algorithms in processor simulators should be explored. Future work may include exploring how varying levels of cache hierarchy,

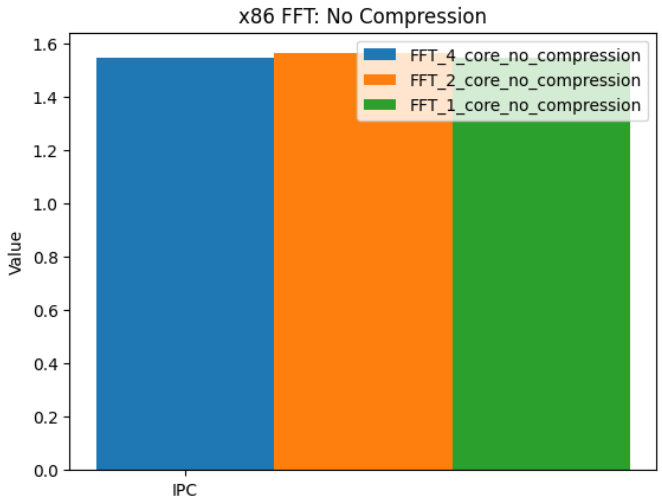

Fig. 8: IPC of x86 FFT simulation with uncompressed L3 cache

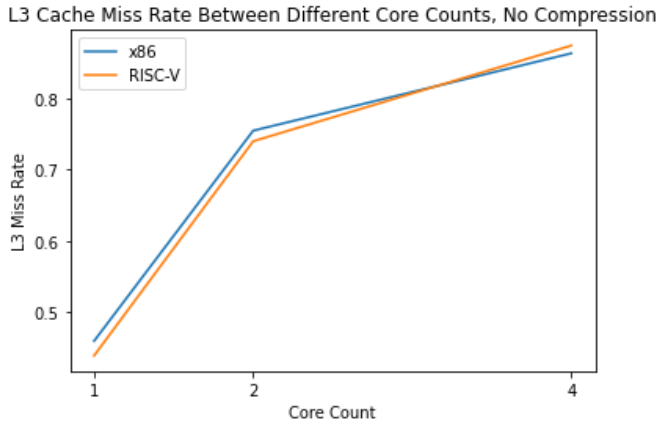

Fig. 9: Cache miss rate for various core counts with no cache compression

mixing different compression algorithms on different levels of the cache, and various cache sizes affect performance. Lastly, since this project only covered three different workloads, more research should be done on the performance differences between multi-core and single-core processors from cache compression with more workloads. As this project consists of analyses of the provided compression algorithms in gem5, more investigation should be done in finding a workload that does not benefit from the given compression algorithms, and developing a compression algorithm that addresses said workload. Further investigation into partially compressed caches could also be researched, specifically looking into testing various compression algorithms for the compressed partition to find an optimal algorithm, especially for caches other than the LLC.

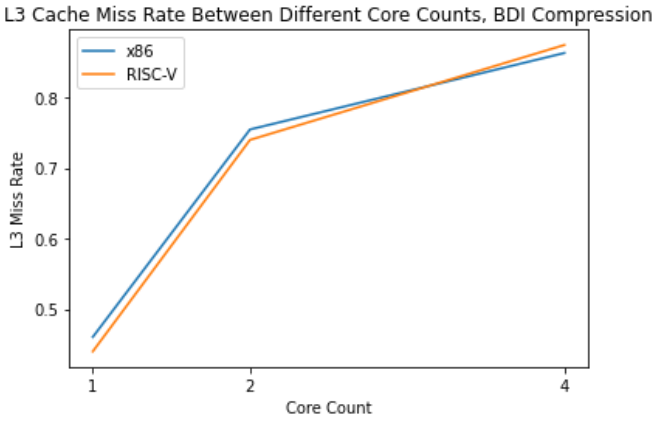

Fig. 10: Cache miss rate for various core counts with BDI compression in L3 cache

References

[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.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011. [Online]. Available: https://doi.org/10.1145/2024716.2024718

[3] D. Chen, E. Peserico, and L. Rudolph, "A dynamically partitionable compressed cache," 2003. [Online]. Available: https://dspace.mit.edu/handle/1721.1/3677

[4] A. R. Alameldeen and R. Agarwal, "Opportunistic compression for direct-mapped dram caches," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–136. [Online]. Available: https://doi.org/10.1145/3240302.3240429

[5] A. R. Alameldeen and D. A. Wood, "Interactions between compression and prefetching in chip multiprocessors," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 228–239.

[6] S. Singh, J.-M. Cheng, B. C. Beardsley, D. P. Leabo, F. L. Wade, M. T. Benhase, and M. E. Goldfeder, "Data caching with a partially compressed cache," Patent, Nov, 2001. [Online]. Available: https://patents.google.com/patent/US6324621B2

[7] D. R. Carvalho and A. Seznec, "Understanding cache compression," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 3, jun 2021. [Online]. Available: https://doi.org/10.1145/3457207

[8] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 18, no. 8, pp. 1196–1208, 2010.

[9] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, ser. MICRO 33. New York, NY, USA: Association for Computing Machinery, 2000, p. 214–220. [Online]. Available: https://doi.org/10.1145/360128.360150

[10] A. Arelakis, F. Dahlgren, and P. Stenstrom, "Hycomp: A hybrid cache compression method for selection of data-type-specific compression methods," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: Association for Computing Machinery, 2015, p. 38–49. [Online]. Available: https://doi.org/10.1145/2830772.2830823

[11] S. Sardashti, A. Seznec, and D. A. Wood, "Yet another compressed cache: A low-cost yet effective compressed cache," *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, sep 2016. [Online]. Available: https://doi.org/10.1145/2976740

[12] A. Jadidi, M. Arjomand, M. T. Kandemir, and C. R. Das, "Hybrid-comp: A criticality-aware compressed last-level cache," in *2018 19th International Symposium on Quality Electronic Design (ISQED)*, 2018, pp. 25–30.

[13] J. Gaur, A. R. Alameldeen, and S. Subramoney, "Base-victim compression: An opportunistic cache compression architecture," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 317–328. [Online]. Available: https://doi.org/10.1109/ISCA.2016.36

[14] C.-C. Huang and V. Nagarajan, "Increasing cache capacity via critical-words-only cache," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, 2014, pp. 125–132.

[15] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 212, mar 2004. [Online]. Available: https://doi.org/10.1145/1028176.1006719

[16] S. Rea and E. Atoofian, "Mitigating critical path decompression latency in compressed l1 data caches via prefetching," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 694–701.

[17] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "An on-chip cache compression technique to reduce decompression overhead and design complexity," *Journal of Systems Architecture*, vol. 46, no. 15, pp. 1365–1382, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762100000308

[18] J. Bucek, K.-D. Lange, and J. v. Kistowski, "Spec cpu2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 41–42. [Online]. Available: https://doi.org/10.1145/3185768.3185771