Exploring the Impacts of Software Cache Configuration for In-line Compressed Arrays

Sansriti Ranjan, Dakota Fulp, and Jon C. Calhoun
Holcombe Department of Electrical and Computing Engineering, *Clemson University* Clemson, USA
Email: { sansrir, dakotaf, jonccal}@clemson.edu

***Abstract*—In order to compute on or analyze large data sets, applications need access to large amounts of memory. To increase the amount of physical memory requires costly hardware upgrades. Compressing large arrays stored in an application's memory does not require hardware upgrades, while enabling the appearance of more physical memory. In-line compressed arrays compress and decompress data needed by the application as it moves in and out of it's working set that resides in main memory. Naive compressed arrays require a compression or decompression operation for each store or load, respectively, which significantly hurts performance. Caching decompressed values in a software managed cache limits the number of compression/decompression operations, improving performance. The structure of the software cache impacts the performance of the application. In this paper, we build and utilize a compression cache simulator to analyze and simulate various cache configurations for an application. Our simulator is able to leverage and model the multidimensional nature of high-performance computing (HPC) data and compressors. We evaluate both direct-mapped and set-associative caches on five HPC kernels. Finally, we construct a performance model to explore runtime impacts of cache configurations. Results show that cache policy tuning by increasing the block size, associativity and cache size improves the hit rate significantly for all applications. Incorporating dimensionality further improves locality and hit rate, achieving speedup in the performance of an application by up to 28.25%.**

***Index Terms*—in-line compression, compressed arrays, lossy compression, big-data, software cache, cache configuration**

I. Introduction

Large-scale high-performance computing (HPC) applications generate large amounts of output data [1], [2]. To analyze the output, applications need large amounts of DRAM. Analyzing a large run on a few nodes or running a large run on a small system is infeasible due to not enough on-node physical memory to process the data. Using more nodes or upgrading the amount of physical memory per-node is costly.

Compression helps to reduce the output storage requirements for an application and can help shrink the memory footprint of a running application [3]. Integrating compression inside an application to shrink the application's footprint is called in-line compression. A compressed array/vector is used exactly like a standard array/vector. However, when the application needs to access a data value from a compressed array, the data must first be decompressed. Similarly, when we need to write to a compressed array, the data must be compressed before the store occurs. Naive in-line compression requires a compression/decompression operation for each memory access, hurting the performance of the application. Moreover, depending on how the array is compressed, each decompression operation might need to fully decompress the array to load a single value. Repeated requests for the same data or data from a nearby element results in additional decompression operations.

Caching the decompressed values in a software managed cache limits the number of compression and decompression operations. The configuration of this cache can significantly impact application performance [4]. As applications differ in their access patterns, the effectiveness of a cache configuration changes. As the need for in-line compression grows, configuring the software cache involves turning several parameters through trial-and-error or not at all.

To aid in configuring in-line compressed array caches, this paper presents a tool to profile the memory access pattern of an application and to simulate various cache configurations to determine the best configuration. Our tool works for both lossy and lossless compressed arrays that use caching. Moreover, our tool leverages common HPC compressor design philosophies such as data dimensionality and decomposing a dataset into small fix-sized blocks.

Specifically, we make the following contributions:

- Construct a cache simulator that predicts the performance of in-line lossy and lossless compressed arrays configurations.
- Devise a performance model to predict the expected execution time of an application.
- Incorporate dimensionality of the array and the decomposition method of the compressor to reduce the number of decompression operations on the critical path and improve the spatial locality of in-line compressed arrays.
- Demonstrate how cache parameterization and access patterns impacts the performance of the application.

The rest of the paper is as follows. In Section II, we describe the background and related work. Section III describes our methodology and design of our cache simulator. It also explains the design decisions and modeling undertaken for the simulator. Section IV describes the experimental setup and results. Finally, Section V summarizes the results and describes our future work.

II. Background and Related Work

A. Caching

Caching is a method for reducing the impact of data movement by keeping data needed for the computation in fast memory. Previous work explores the use of hardware compression to expand the size of hardware caches and main-memory [5]–[12] or caching at the software level [13]–[16]. Software caches have also been used to improve I/O performance for distributed applications [17], [18], and to cache the input data for parallel tasks [19], [20]. Two popular types of cache are:

Direct Mapped Cache: Cache where each address maps to exactly one location. To determine if a value is present, the address is hashed, yielding a location inside the cache. Provided the entry is valid and the address tag matches, the value is found in the cache. Storing a single value in each cache block enables many dispirit addresses to be stored in the cache. However, such a design does not leverage spatial locality. To improve spatial locality, the size of each cache block can grow to hold several spatially close addresses. The simple mapping logic enables fast access, but as the block size grows, fewer dispirit addresses are present, which can lower temporal locality. Moreover, if two addresses map to the same cache location, an eviction occurs and the evicted block is replaced by a new block containing the address of the load/store.

Set Associative Cache: Improves over a direct mapped cache by allowing multiple cache blocks at each location (set). A k-way set associative cache has k blocks in each set. To determine if a value is present in the cache, the address is partitioned into 3 bit fields: *tag*, *index*, and *offset*. The *index* determines what set the address maps to. The tag along with a dirty bit is used to determine if the address is within any block in the set. Finally, the *offset* is used to locate the address location inside the identified block.

B. Compressors

While many lossy compression frameworks do not enable random access into the compressed data, one notable exception is ZFP's fixed-rate error-bounding mode [21]. ZFP's fixed-rate mode splits the data into 4^d sized blocks, where d is the dimensionality of the data, and compresses each block individually to achieve a user-specified desired bit-rate. Using ZFP compressed arrays enables the application to compute on datasets that exceed the capacity of main-memory.

This approach enables users to access individual data values without decompressing the entire data, only the 4^d block containing the values. As decompressing and compressing blocks for each read and write operation is cost-prohibitive, this framework also includes a small direct-mapped software cache that stores decompressed data blocks. This cache uses a write-back policy with a dirty bit stored with each cache block to avoid unnecessary recompression of data blocks that the application has not altered. Not recompressing unaltered data blocks saves time and ensures the data fidelity is not impacted unnecessarily. While their cache size is configurable, the default size accommodates two layers of data blocks for the x-dimension, which the ZFP developers found to yield high-performance. Prior work shows that tuning the parameters of ZFP's software cache yield a performance improvement of up to 8% [4].

Other compressors such as SZ [22] have a random access mode but do not support caching. Moreover, any lossy or lossless compressor has random access capability if the data is first "chunked" and the compressor applied to each chuck independently. Integrating a software cache into these compressors will make them more amenable for in-line compressed arrays. The cache simulator we present in this paper is useful for exploring performance aspects related to the integration.

III. In-line Compressed Arrays Cache Simulator

In-line compressed arrays compresses and decompresses data when needed, giving the appearance of larger memory capacity. If a cache is present, it is on the critical path; therefore, configuration is important. Larger caches have larger search times. Prior work shows that the cache's configuration can improve performance by up to 8% for matrix-matrix multiplication [4]. To improve the performance of in-line compression, decompressed values can be stored in a software managed cache. When configuring the cache one must set several parameters such as total size, block size, replacement policy, associativity level.

Traditional hardware caches have a linear view of main memory and only exploit spatial locality in a single dimension through the memory addresses. When operating on an array that is compressed, the compressor knows additional pieces of information about the array (e.g. dimensionality, data type). This additional information is leveraged during compression to yield higher levels of reduction. For example, both SZ [22] and ZFP [21], two popular HPC lossy compressors, decompose multidimensional data into small multidimensional blocks (e.g. ZFP block size is 4^d, where d is the dimensionality) and processes each block individually. If the compressed array's cache is not aware of this decomposition, then performance suffers as only a subset of each compressor's block is placed in each of the cache's block. The remaining decompressed values are discarded. To finish filling the cache block, additional decompressions are necessary. Figure 1 illustrates this limitation. Moreover, for certain access patterns (e.g. walking down a column), this leads to additional decompressions, negating the benefits of caching. If we leverage multidimensional caching and map the full compressor block into a cache block, we only require one decompression for each cache miss and improve spatial locality for certain access patterns.

To more easily configure caches in compressed arrays, we develop a cache simulator that takes in an address trace from an application and processes the sequence of loads/stores on various cache configurations that incorporates knowledge from the compressors to determine what configuration yields

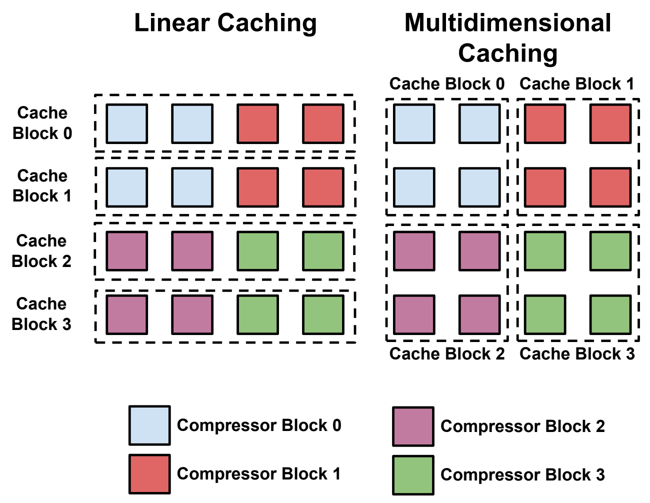


Fig. 1: Caching of a 4×4 array. Each cache block (dashed lines) houses 4 elements of the array. The compressor decomposes the data into 2×2 blocks.

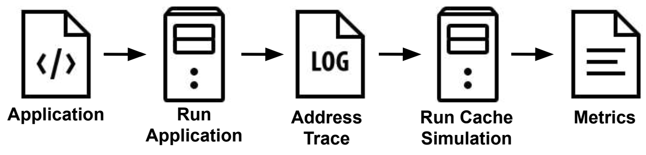


Fig. 2: Overview of our cache simulator workflow.

the best performance. Figure 2 shows an overview of this workflow.

A. Address Trace Generation

As HPC applications progress, they read and write data at various memory locations. HPC applications differ in their access patterns, which in turn can lead to performance variability. To ensure that we select the best cache for an application and to avoid the need to run an application multiple times, we extract an address trace for the variables allocated as compressed arrays. We accomplish this via instrumenting the code to first log the base address of each in-line compressed array along with the data type, dimensionality, and compressor properties such as block size. Then, for each memory access, we log the tuple: (load/store, variable name, address)[1]. In the tuple, the address logged is that of the array element loaded. When the base address and the dimensions of the array are run through our cache simulator, we transform the linear memory address into a multidimensional index. The multidimensional index enables us to determine what compression block the array element maps to. Alternatively, at the expense of larger address trace files, we record the array indices in the tuple directly to reduce the computational requirements of the simulator. Having the address trace allows us to readily parallelize the evaluation process for each cache configuration.

[1]We do not record the value loaded as it has no bearing on the sequence of loads/stores.

B. Cache Simulator

Given an address trace from an application, our cache simulator replays the sequence of loads/stores and quantifies the expected performance. Because an application may contain multiple compressed arrays, each with different access patterns, we explore two types of caching for compressed arrays. The first involves a unified cache that is shared between multiple compressed arrays. The second is a dedicated cache for each compressed array.

The first approach limits the memory overhead of caching. This approach is useful for applications that operate on one or a few compressed arrays at a time. For a fixed size cache, as the number of compressed arrays in the working set increases, so too does the likelihood for conflicts and cache misses. The second approach is designed for situations where multiple compressed arrays are used at the same time. With each compressed array having its own private cache, the likelihood of conflicts inside the caches diminishes. Moreover, the configuration of each cache is tunable.

To configure each cache, our cache simulator needs several parameters. The first four parameters are: the data type of the array's elements, the cache's size (bytes), the size of each cache block (bytes), and the set associativity level of the cache[2]. For compressors that decompose the array into blocks when compressing, the cache's block size should mirror that of the compressor and the cache's blocks need to be multidimensional (see Figure 1). Setting the block size this way enables the cache block to house all the elements decompressed at the same time, providing spatial locality in multiple dimensions, which results in fewer decompression operations and improves performance.

After the block size is defined, the number of sets in the cache is computed by dividing the cache size by the size of each set (cache block size multiplied by the associativity level). Figure 3 shows an example of direct-mapped and a k-way set associative cache. Finally, the cache's replacement policy is set. For this work, we use *least recently used (LRU)*, but other policy can be leveraged.

Once all the caches are constructed, the simulator processes the address trace. Each load and store is directed to the appropriate cache. From the address, we determine the compressor block in which the address resides. That block information determines which set inside the cache to search in. For each block inside the identified set, we attempt to match the tag. For univariate caches, the tag is the block ID. For multivariate caches, the tag is the block ID and an ID corresponding to the allocation. If the block is found, a *hit* occurs. However, if the block is not found, a *miss* occurs. The compressor block containing the needed value must be decompressed and placed into the cache. For the block that is evicted from the cache, compression occurs before it is stored to main-memory.

[2]For a direct-mapped cache, the set associativity level is $k = 1$.

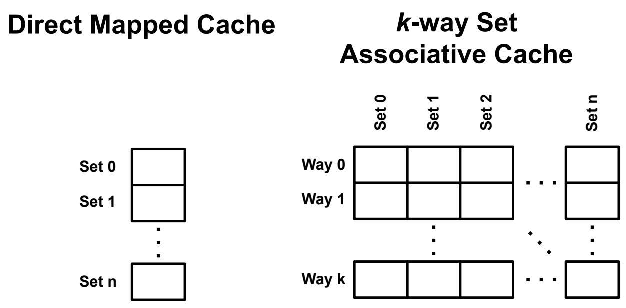


Fig. 3: Diagram of direct-mapped and a k-way set associative cache.

However, since our simulator is data and compressor agnostic, we model the cost of compression and decompression (see Section III-C).

C. Performance Model

As the address trace is being processed via our cache simulator, the simulator computes various performance metrics. We use these performance metrics in Section IV to determine how cache parameterization impacts performance across several HPC kernels.

The two primary metrics that are collected are the number of cache hits, $nHit$, and cache misses, $nMiss$. From these values, we compute the hit-rate, R_H, and miss-rate, R_M, as follows:

$$R_H = \frac{nHit}{nHit + nMiss} \tag{1}$$

$$R_M = 1 - R_H \tag{2}$$

Each hit results in fast retrieval of the requested datum. However, when a miss occurs, the block containing the requested value is decompressed and placed into the cache and the evicted block is compressed and written back. Thus, having a high hit-rate is indicative of good performance.

To better understand the performance differences between configurations, we compute the time cost for processing the trace. We model the time for a hit in the cache as:

$$T_H = t_{id} + t_{search} + t_{fetch}, \tag{3}$$

where t_{id} is the time cost to determine the block ID given an address, t_{search} is the time to search a set, and t_{fetch} is the time to retrieve the datum and return it to the user. The computation for each is rather small, with the t_{id} calculation taking the longest, due to several integer arithmetic operations. The searching cost, t_{search}, involves a comparison of the tag, and grows as the associativity level of the cache grows. The fetch time, t_{fetch}, is that of loading a datum from memory into a register.

The cost of a miss is more substantial. A miss contains all the cost of a hit, but also incurs additional costs due to compression and decompression. If spare computational resources are available, compressing the evicted block is removed from the critical path (done asynchronously in the background) as it is not needed to satisfy the outstanding load by the application, leading to the following:

$$T_M = T_H + t_{decompress}. \tag{4}$$

Combining our metrics, we obtain the total time cost of memory operations for the compressed array:

$$T_{tot} = nHit \cdot T_H + nMiss \cdot T_M. \tag{5}$$

For applications making use of multiple caches, we compute the overall application time as the sum of T_{tot} for each cache.

TABLE I: Problem Size for Applications

Name	Problem Size	Number of Loads	Number of Stores
Dijkstra	1000 nodes	1000091	2000941
FFT	10^{10}	3543380	6726400
Jacobi	1000×1000	24928052	32400013
Matmul	1000×1000	2502500	7497500
SpMV	1000×1000	1602296	1901001

IV. Experimental Results

A. Testing Methodology

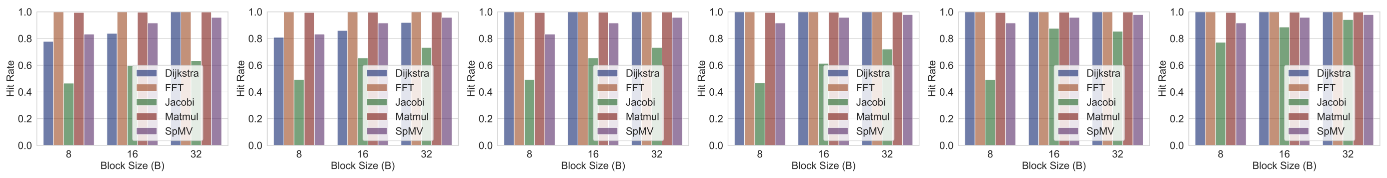
To explore the performance impact of cache configuration on compressed arrays, we run five HPC kernels detailed in Table I. Experiments are run on Clemson's Palmetto Cluster, and we test direct-mapped, 2-way set associative, and 4-way set associative caches. Cache sizes range from 0.01 MB to 0.1 MB and block sizes vary between 8 B and 32 B.

B. Impact of Block Size and Set-Associativity in a 1D Cache Configuration

The five HPC kernels are first run with different cache configurations where the cache size, block size and associativity of the cache is varied. The kernels are run with the dimensionality of the cache as linear and a dedicated cache for each compressed array.

Figure 4(a)–(c) shows the hit rate vs the block size for the five kernels for a cache size of 0.01 MB. We see that when increasing the block size for all values of k, the hit rate increases. This is because as the size of a single block increases, more elements of the array are stored in it, improving the spatial locality and the hit rate. FFT and Matmul have a hit rate which is very high (0.99) while Jacobi has a significantly low rate (0.67-0.82).

While both Matmul and Jacobi have the same problem size, but Jacobi has many more loads/stores due to its iterative nature, allowing it to run for a user defined iteration number before terminating. Matmul performs better than Jacobi due to the difference in the access patterns between the two applications. While Matmul accesses each matrix either by row or by column, Jacobi's pattern is based on its stencil. For



(a) Direct Mapped. (b) Two-way Set Associative. (c) Four-way Set Associative. (d) Direct Mapped. (e) Two-way Set Associative. (f) Four-way Set Associative.

Fig. 4: Impact of Block Size and Associtivity on 1D Caching. (a)–(c) uses a cache size of 0.01MB. (d)–(f) uses a cache size of 0.1MB.

example, the 5-point stencil requires data from three rows: 1 element from the previous row, 3 elements from the current row, and 1 element from the next row. Needing three rows of values means that the probability that all the elements to be accessed are present in the cache is lower for the same cache size. Thus, the lower hit rate. However, increasing the block size increases the number of elements present in the cache block by the same factor, improving spatial locality.

Increasing the associativity increases the number of cache blocks within the same set, and decreases the chance for evictions due to aliasing. Furthermore, the probability that all elements to be accessed are present in the cache increases, leading to increases in the hit rate evident in Figure 4(b)–(c). Yet, the increase in hit rate of Jacobi compared to Matmul is low due to Jacobi's access pattern. For all the applications, by increasing the cache size to include a greater number of cache blocks, the hit rate improves, as we see in Figure 4(d)–(f).

C. Sensitivity of 2D Cache Configuration

All five kernels except SpMV can utilize multidimensional caching. SpMV has linear accesses throughout. This is because our SpMV kernel uses co-ordinate format (COO) to represent the sparse matrix as three 1D vectors of the row index, column index, and value of each of the non-zero entries. Thus, all the arrays for the kernel are 1D and are not able to take advantage of multidimensional caching.

For the remaining kernels, the 2D cache configuration is able to leverage the access patterns and spatial locality to improve performance. For example, Matmul multiples two matrices A and B together. A is accessed row-wise (elements are adjacent in memory and yield a high hit rate). B is accessed column-wise (elements are nonadjacent in memory and yields a low hit-rate with 1D). However, with the 2D configuration, multiple column entries are brought into the cache, improving the hit rate. Moreover, using a dedicated cache approach is useful to ensure the appropriate locality is maximized, improving the hit rate.

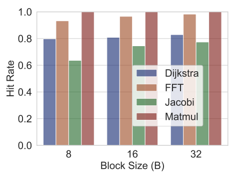
The four remaining kernels are first run to generate the address trace where the array indexes are configured in 2D. After address trace generation, the trace is run with the simulator where the cache size, block size and associativity is tuned based on the testing methodology. Figure 5 shows that increasing the block size from 8 to 32 improves the hit rate similarly to the 1D cache configuration (see Figure 4). Furthermore, increasing the associativity further increases the hit rate as the number of blocks in each set increases, reducing the evictions due to aliasing. FFT and Matmul do not show any significant improvement in hit rate as the cache size and 2D configuration is sufficient for their respective problem sizes, leading to a 0.99 hit rate. Dijkstra and Jacobi improve significantly on increasing the associativity. Increasing the cache size yields a very high hit rate in all the kernels because the larger cache size is able to house more of the array elements. Again, as associativity level increases, aliasing evictions decreases.

The most important point to note here is that in going from 1D Caching to 2D Caching, all the applications improve in their hit rate. Multidimensional caching therefore enables faster access to the data, which helps improve the performance of the application. This is attributed to the speedup achieved by having the block of array elements easily accessible in the cache in its decompressed form (as the blocks are accessed in the same manner as the compressor). We explore the performance implications of multidimensional caching in Section IV-D.

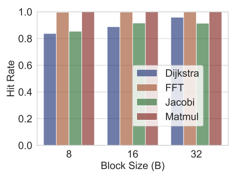
D. Performance Implications

From the experimental results of the cache simulator, tuning the parameters of a cache and its dimensionality improves the hit rate for all applications when the application is run with inline compression. Yet, to understand how this caching transitions into an improvement in performance of the application, the time taken with and without multidimensional caching is computed. Using the performance model from Section III-C, we compute the time taken for all combinations of cache size, block size, associativity, and dimensionality.

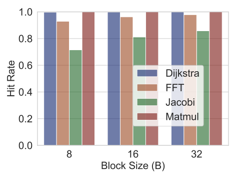
Figure 6 displays the speedup achieved over 1D caching for each of the applications based on the metrics hit rate, miss rate and the cache parameters. For this performance model, we select a cache size of 0.1MB and a block size of 32. We choose this configuration because it gave the best performance across all applications. It is evident that there is considerable speedup for all the applications when using a multidimensional cache. Dijkstra shows the highest speedup with 28.25%, followed by Matmul (24.17%), Jacobi (12.95%), and FFT (12%). These improvements are in accordance with the hit rate improvement seen in 2D caching for these application in Figures 5(d)–(f). Jacobi shows a considerable amount of speedup based on its access pattern, needing data from multiple rows and columns. 2D caching results in increased spatial locality and an increase in hit rate, enabling it to reduce the time taken to run.



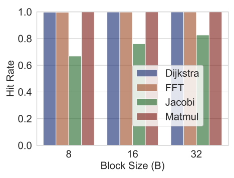
(a) Direct Mapped.



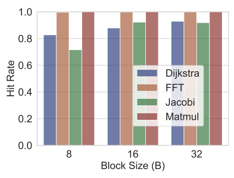
(b) Two-way Set Associative.



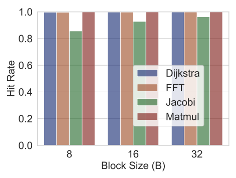
(c) Four-way Set Associative.



(d) Direct Mapped.



(e) Two-way Set Associative.



(f) Four-way Set Associative.

Fig. 5: Impact of 2D Caching. (a)–(c) uses a cache size of 0.01MB. (d)–(f) uses a cache size of 0.1MB.

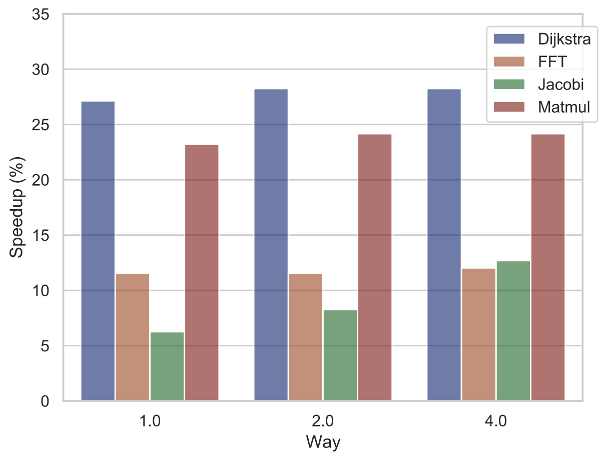


Fig. 6: Application speedup with 2D caching compared to 1D caching.

E. *Unified Versus Individual Caches*

Table II compares the hit rate between a unified cache approach for each application and an approach where there is an individual cache for each of the compressed arrays. We parameterize the performance model with a cache size of 0.1 MB, block size of 32 and four-way associativity for a 2D cache except for SpMV which uses 1D caching. We select this configuration because it gives the best performance across all applications. Investigating how each application performs, we see that for all applications, using an individual cache always gives better performance. This is because having individual caches enables these applications to have a higher hit rate by increasing the probability of data to be accessed already present in the cache. From the table we see that Dijkstra, FFT and SpMV comparatively benefit from individual caches while Matmul and Jacobi show the same hit rate for both approaches and can utilize both the approaches. Applications which utilize multiple compressed arrays simultaneously need an individual cache for each array because using a unified cache increases collisions. However, applications that operate on a single array at a time can utilize a unified cache without the same performance degradation.

Overall, the cache simulator helps understand not only the improvement in performance of an application by tuning different parameters, but also which cache configuration and cache approach is optimal for a specific application.

TABLE II: Comparison of the hit rate for various caching approaches.

Application Name	Hit Rate - Individual	Hit Rate - Unified
Dijkstra	0.98	0.94
FFT	0.99	0.88
Jacobi	0.92	0.92
Matmul	0.99	0.99
SpMV	0.97	0.95

V. Conclusion

Current software caches linearize arrays and do not incorporate the dimensionality of an application. We build a cache simulator which incorporates dimensionality along with tuning other parameters. The results show that increasing the cache size improves the hit rate and performance of an application, provided it has high spatial locality and no random access patterns. Results further show that increasing the associativity or cache policy improves the hit rate further and can add to an improvement in the performance for applications with low spatial locality. Based on the application's access patterns and locality, the simulator helps to configure an optimum cache and gauge the improvement in performance of the application. It helps to assess the trade-off between cache size, block size and cache policy for applications with low spatial locality. Moreover, adapting certain applications to 2D can improve the performance of the application. Future work involves how to optimize cache configurations for applications with low spatial locality and random access patterns.

Acknowledgments

Clemson University is acknowledged for generous allotment of compute time on the Palmetto cluster. This material is based upon work supported by the National Science Foundation under Grant No. SHF-1910197 and SHF-1943114.

References

[1] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, K. Heitmann, K. Kumaran, V. Vishwanath, T. Peterka, J. Insley *et al.*, "HACC: Extreme scaling and performance across diverse architectures," *Communications of the ACM*, vol. 60, no. 1, pp. 97–104, 2016.

[2] NYX simulation, https://amrex-astro.github.io/Nyx, 2019, online.

[3] F. Cappello, S. Di, S. Li, X. Liang, A. M. Gok, D. Tao, C. H. Yoon, X.-C. Wu, Y. Alexeev, and F. T. Chong, "Use cases of lossy compression for floating-point data in scientific data sets," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1201–1220, 2019.

[4] P. Triantafyllides and J. C. Calhoun, “Analyzing the performance of zfp compressed arrays on hpc kernels,” in *Poster Session of the 2019 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’19. Washington, DC, USA: IEEE Computer Society, 2019.

[5] S. Sardashti and D. A. Wood, “Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching,” in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 62–73. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540715

[6] S. Sardashti, A. Seznec, and D. A. Wood, “Skewed compressed caches,” in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 331–342. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.41

[7] ——, “Yet another compressed cache: A low-cost yet effective compressed cache,” *ACM Trans. Archit. Code Optim.*, vol. 13, no. 3, pp. 27:1–27:25, Sep. 2016. [Online]. Available: http://doi.acm.org/10.1145/2976740

[8] 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: ACM, 2015, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830823

[9] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, “Base-delta-immediate compression: Practical data compression for on-chip caches,” in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’12. New York, NY, USA: ACM, 2012, pp. 377–388. [Online]. Available: http://doi.acm.org/10.1145/2370816.2370870

[10] X. Chen, L. Yang, R. P. Dick, L. Shang, and H. Lekatsas, “C-pack: A high-performance microprocessor cache compression algorithm,” *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 18, no. 8, pp. 1196–1208, Aug. 2010. [Online]. Available: http://dx.doi.org/10.1109/TVLSI.2009.2020989

[11] J. S. Miguel, J. Albericio, A. Moshovos, and N. E. Jerger, “DoppelgÄnger: A cache for approximate computing,” in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 50–61. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830790

[12] A. Jain, P. Hill, S. Lin, M. Khan, M. E. Haque, M. A. Laurenzano, S. A. Mahlke, L. Tang, and J. Mars, “Concise loads and stores: The case for an asymmetric compute-memory architecture for approximation,” in *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*, 2016, pp. 1–13. [Online]. Available: https://doi.org/10.1109/MICRO.2016.7783744

[13] S. Perarnau, J. A. Zounmevo, B. Gerofi, K. Iskra, and P. Beckman, “Exploring data migration for future deep-memory many-core systems,” in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2016, pp. 289–297.

[14] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale, “Di-mmap—a scalable memory-map runtime for out-of-core data-intensive applications,” *Cluster Computing*, vol. 18, no. 1, pp. 15–28, Mar 2015. [Online]. Available: https://doi.org/10.1007/s10586-013-0309-0

[15] A. Shafiee, M. Taassori, R. Balasubramonian, and A. Davis, “Memzip: Exploring unconventional benefits from memory compression,” in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2014, pp. 638–649.

[16] J. Li, F. Zafari, D. Towsley, K. K. Leung, and A. Swami, “Joint data compression and caching: Approaching optimality with guarantees,” in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’18. New York, NY, USA: ACM, 2018, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/3184407.3184410

[17] N. S. Islam, X. Lu, M. Wasi-ur-Rahman, R. Rajachandrasekar, and D. K. D. K. Panda, “In-memory i/o and replication for hdfs with memcached: Early experiences,” in *2014 IEEE International Conference on Big Data (Big Data)*, Oct 2014, pp. 213–218.

[18] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt, “Integrating online compression to accelerate large-scale data analytics applications,” in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 1205–1216.

[19] Y. Wang, R. Goldstone, W. Yu, and T. Wang, “Characterization and optimization of memory-resident mapreduce on hpc systems,” in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, May 2014, pp. 799–808.

[20] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, “Pacman: Coordinated memory caching for parallel jobs,” in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 267–280. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/ananthanarayanan

[21] P. Lindstrom, “Fixed-rate compressed floating-point arrays,” *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2674–2683, 2014.

[22] X. Liang, S. Di, D. Tao, S. Li, S. Li, H. Guo, Z. Chen, and F. Cappello, “Error-controlled lossy compression optimized for high compression ratios of scientific datasets,” in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 438–447.