Coherence Traffic and Store-Value Similarity Characterization for Approximate Workloads

***Abstract*—Parallel and shared memory applications are becoming increasingly important to fully utilize the multi-core architectures present in modern computing systems. The key to improving these systems lies within the on-chip networks connecting cores and memories. Efficient communication between nodes in the network is necessary to see further performance gains; and the primary source of communication traffic emanates from cache coherence. In this paper we characterize error-tolerant multi-core workloads in terms of generated coherence traffic and store-value similarity. In our work, store-value similarity is defined as a distribution of the relative difference between a store value and the value being overwritten in the cache line. The greater the store-value similarity, the smaller the relative differences of the stored values. As many applications are intrinsically error-tolerant, for example those who's computation use floating point data, we can gain insight from the collected metrics to reduce network coherence traffic by executing on approximate/incoherent values. We also explore the domain of approximate computing and propose a novel coherence optimization, Silent Approximate Stores (SAS). In SAS, upon a store hit, if the store value is approximately equal (determined by an application define error tolerance) to the value being overwritten in the cache line, coherence requests to other nodes will not be sent. The value is stored as is maintaining the current coherence state of the line, in return mitigating further network traffic from cache misses due to coherence invalidations.**

***Index Terms*—blah.....**

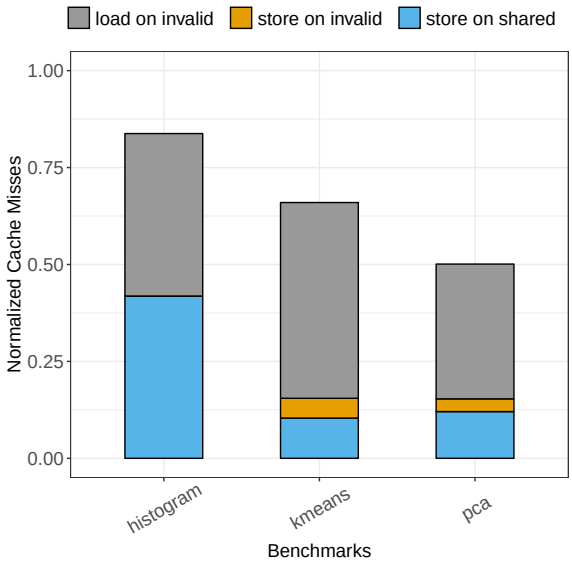


Fig. 1: Breakdown of L1-d cache misses due to coherence states, i.e., invalid or read-only (shared). All other misses are due to lines not being in cache.

I. Introduction

Uni-core processor scaling over that past decade has stagnated due to increasing power consumption for marginal performance gains. Modern computing systems are seeing proliferation into multi-core and many-core processors. Parallel and shared memory applications are becoming increasingly important to fully utilize these current and upcoming architectures. However, the communication bandwidth between compute units is the main bottleneck for further boosts in performance. The key to future advancements lies within the communication infrastructure, specifically the on-chip networks (NoCs) connecting cores and memories. NoC traffic congestion stems from intensive memory accesses issued from the on-chip cores [10], and with multiple cores using a shared memory model, theses accesses are largely made of cache coherence traffic.

Memory access misses private caches are either due to the cache line not being present or in the wrong coherence states. In a write-invalidate cache coherence protocol, a store operation to a local cache which misses for any reason would send invalidation requests to remote caches with the same cache line. Subsequent load or stores in the remote caches to that line would register a miss and generate corresponding coherence requests, i.e., traffic in the NoC. Figure 1 shows a breakdown of L1-d cache misses normalized to the total amount of cache misses, on three benchmarks (Described in Section V). At least 50% of the misses are due to being in the wrong coherence states, i.e., loads and stores on coherence invalidated lines, or stores on read-only (shared) lines. All other misses are due to the line not being present in the cache. In our preliminary findings, a substantial portion of misses are on loads due to invalidates caused by stores (put average number here), with a non-trivial portion of the coherence misses are on stores (put in average number here).

A significant portion misses due to coherence related invalidation come from silent stores [4], storing values that exactly match what is being overwritten. This suggest that there is a high degree of value locality [5]. Existing work has examined squashing silent stores for reduce misses and improve performance [3, 4]. We however delve into the domain of approximate computing to expolit the intrinsic error-tolerance of certain applications to minimize coherence traffic with "acceptable" loss in application output quality. We do so by first introducting a novel workload characterization metric – *store-value similarity*. *Store-value similarity* describes a probability distribution of the relative difference of a store value, and the value being overwritten in a memory location. It can be used

to classify which applications would benefit from computing on approximate values to trade off output quality with a reduction in NoC traffic. Higher *store-value similarity* corresponds to a higher probabilty of stores overwriting a memory location with values in a narrower range. Next we propose a coherence protocol optimization, Silent Approximate Stores (*SAS*) tailored for error-tolerant, approximate applications to reduce coherence traffic by exploiting stores with high *store-value similarity*. The protocol implements *approximate stores*, where if the relative difference between a store value and what is being overwritten is within a programmer defined tolerance, the store is completed maintaining the lines current coherence state. This minimizes coherence traffic by inhibiting invalidations to remote caches which in turn reduces the amount of misses on subsequent accesses to invalidated lines.

To summerize, our work contributes two novel ideas from the approximate computing paradigm to exploit inherent error-tolerance in applications amenable to approximation.

- We define a new characterization for approximate applications – *store-value similarity*, a metric to estimate the influence of approximations in computation to the reduction of coherence traffic.
- We propose a coherence optimization, Silent Approximate Stores (*SAS*) to reduce coherence traffic in NoCs.

Our preliminary implementation shows... TODO after evaluation, put some data here...

II. Silent Approximate Stores Protocol

Make a MESI coherence diagram showing how *approximate stores* loop back into current state.

Explain each *approx store* transitions

Maybe explain the two variants. One where you actually store the value, and one where you don't.

III. Related Work

TODO

IV. Implementation

1) gem5 syscall Ruby memory
2) MESI Two-level cache
3) talk about limitations of two-level since now modern procs use 3 levels.
4) talk about increasing simulated L1 to be effectively L2 size to get more accurate results for coherence misses.
5) Possibly talk about Silent Approximate Stores in here

Problem: Successive approximate stores that have values within the error tolerance might grow unbounded and other cache lines in private caches will become very incoherent/stale. E.g., an instruction like `a += 1`.

Solution 1: Broadcast the store value every n cycles, i.e., 100000 cycles.

Solution 2: Approximation is done up the the n-th bit, i.e., the 3rd bit so approximations are bound to be plus/minus 3 from the previous value.

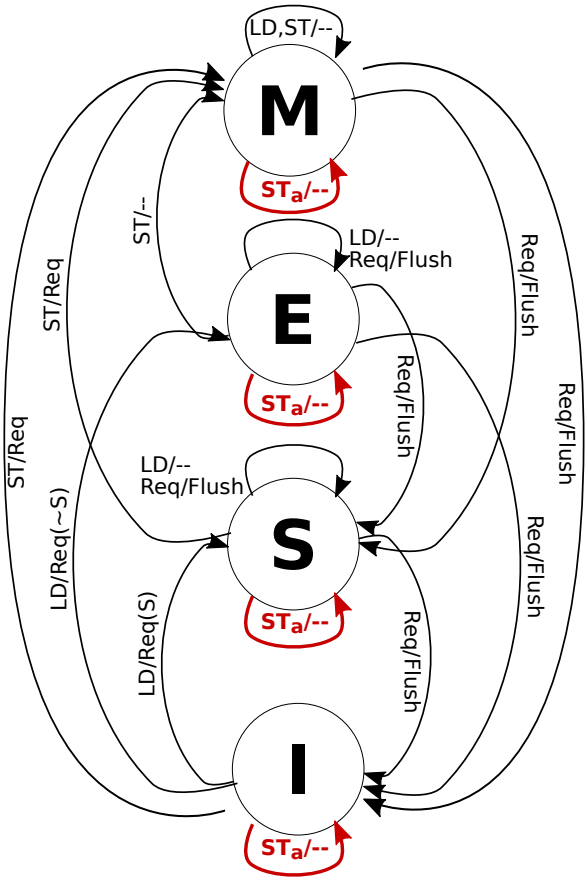


Fig. 2: MESI SAS

V. Evaluation

Benchmarks

We choose five application kernels to characterize from Phoenix – a shared memory implementation of Google's MapReduce model. These workloads represent computations from domains amenable to approximations e.g., image processing and machine learning. The following breifly details each benchmark used in the evaluation.

Histogram counts the frequency of RGB value occurances per pixel from a range of 0-255. The application divides the image into sections and computes histograms on each part seperately. Seperate histograms are reduced into a single, total histogram once all parallel computations are complete. Error-tolerant features of the application can include loading/storing of the pixel data. Small deviations in the frequency of occurances in the range from 0-255 are likely to go unnoticed. For the evaluation we use a 100MB bitmap as the input.

KMeans implements an iteration of the iterative convergence kmeans clustering algorithm. The algorithm consists of a cluster assignment step and an update step. Cluster assignment takes a subset of the input data points and assigns them to appropriate "nearest" clusters, i.e., having the least squared Euclidean distance. The update step calculates the new means of each cluster for the next iteration of the algorithm.

Error-tolerant computions may include the calculations of the Euclidean distance in the cluster assignment step, and the mean for each cluster in the update step. For the evaluation we use an input of 10000 data points.

Linear Regression Something... Input size of 100MB.

Matrix Multiply Something... Multiplies two matrices of size 500x500

PCA implements a segment of the Principal Component Analysis algorithm, namely the mean and covariance calculations. The input is a matrix of pseudo-randomly generated points. The mean calculation paralellizes the generation of the mean vector by computing on subsets of the input matrix rows. The covariance calculates portions of the covariance matrix in parallel given a subset of data. Both mean and convariance computations are amenable to approximation. For the evaluation we use an 1000x1000 matrix as the input.

Simulator

gem5 simulator. The simulated machine is described in Table: I

TABLE I: Simulated Machine Configuration

Parameter	Values
Cores	8 Cores, X86, In-order, Atomic Memory Accesses
L1	Private 64kB DCache/32kB ICache, 2-Way Set Assoc., 64B Block, Pseudo-LRU
L2	Private Unified 1MB, 8-Way Set Assoc., 64B Block, Pseudo-LRU
L3 / LLC	Shared 20MB, 16-Way Set Assoc., 64B Block, Pseudo-LRU
Coherence	Directory MESI Protocol

Coherence Misses

Coherence Misses are defined as misses due to the cache line being in the wrong coherence state, i.e., the tag is found in the cache, but the cache access is a store on a read-only/invalid line, or a load on an invalid line, resulting in a miss.

Observations on the amount of coherence misses seen in Figures 3 and 4.

1) The ratio of cache misses to cache accesses is higher in the L2 cache, many of which are due to coherence misses. Possibly because the L2 cache is larger, and is able to hold more cache lines in read-only/invalid coherence states for stores/loads to miss on.
2) A considerable chunck of the coherence misses (seen in the (c) subfigures) are from stores. Although most of them are from loads, implementing SAS might decrease misses from subsequent loads.
3) Matrix Multiply benchmark performs well in that there are very little cache misses, not just coherence misses. Input matrices are partitioned onto threads with little migrating data (seems to be only a published data type of sharing, one parent and multiply readers) and the computions seem to have good spatial locality which might explain why we see little misses.
4) PCA shows minor amount of misses in the L1, a substantial amount in the L2, neither of which are due to misses on wrong coherence states. This could be due to the calculation of the covariance matrix. The algorithm reads from the matrix of points and a matrix of means to calculate the covariance. Since both matrices are large enough to overflow the L2 and accesses its elements has a long distance in address space, this could be the reason the L2 sees a lot of misses. Not to sure why L1 doesn't generate a lot of misses.
5) Linear Regression sees high amount of coherence traffic, specifically from false sharing due to the `lreg_args` structure. Array `tid_args` of `lreg_args` in the main thread passes each `lreg_args` to the compute threads. However the structures may not be aligned to the cache block which would result in false sharing since the compute threads read and modify `lreg_args` [6, 7].
6) Histogram has similar behaviour to Linear Regression due to a shared heap structure `thread_arg_t`. Each compute thread's `thread_arg_t` structure may not be aligned to the cache line causing false sharing while the threads are reading and modifying [7].

Store-Value Similarity

Benchmark Histogram – shown in Figures 5 and 6

Benchmark KMeans – shown in Figures 7 and 8

Benchmark Linear Regression – shown in Figures 9 and 10

Benchmark Matrix Multiply – shown in Figures 11 and 12

Benchmark PCA – shown in Figure 13. Since there were many unique approximatable addresses to process for this benchmark, processing the results for all of them took a long time. I only show the results for the top 5 traffic generating stores

VI. Conclusion

References

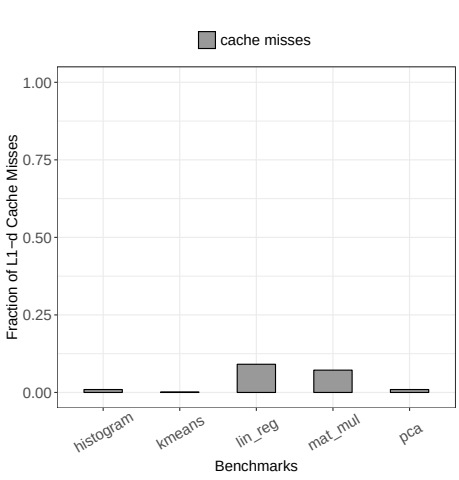
[1] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* (2011).

[2] J. Huh et al. "Coherence Decoupling: Making Use of Incoherence". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2004.

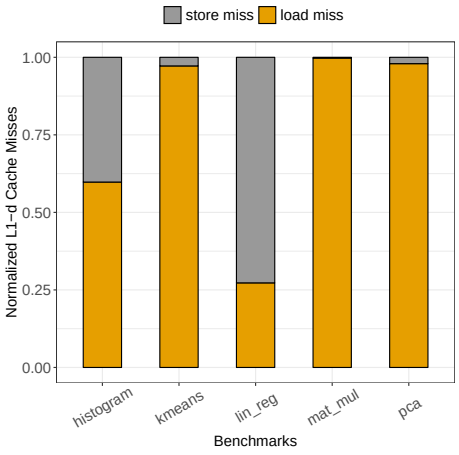
[3] K. M. Lepak and M. H. Lipasti. "Silent stores for free". In: *International Symposium on Microarchitecture (MICRO)*. 2000.

[4] Kevin M. Lepak and Mikko H. Lipasti. "Temporally Silent Stores". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2002.

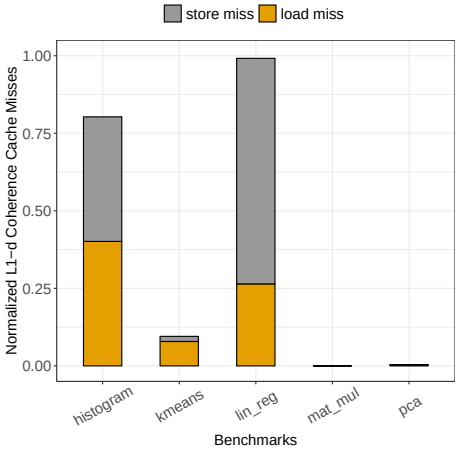
[5] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. "Value Locality and Load Value Prediction". In: *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1996.



(a) Fraction of cache accesses that resulted in misses on the L1-d.

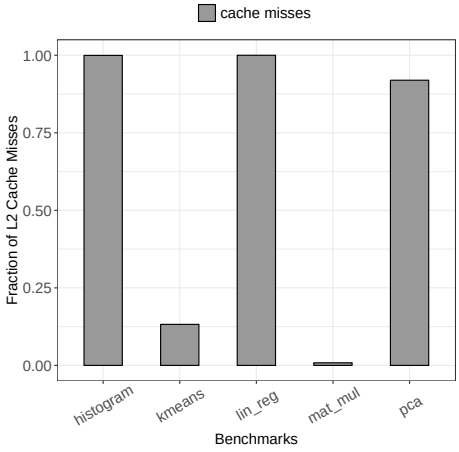


(b) Breakdown of all L1-d cache misses into misses on loads or stores.

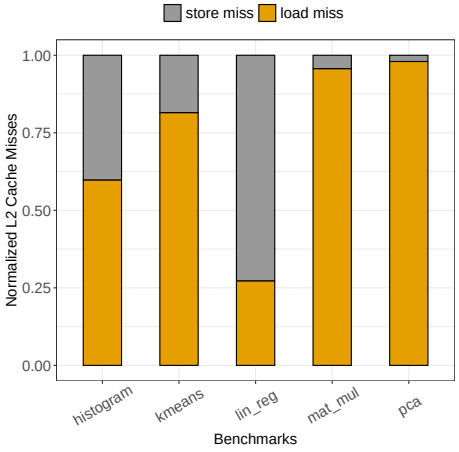


(c) Breakdown of the L1-d cache misses into load misses or store misses due to coherence state. Misses that are not shown are due to capcity and conflict misses.

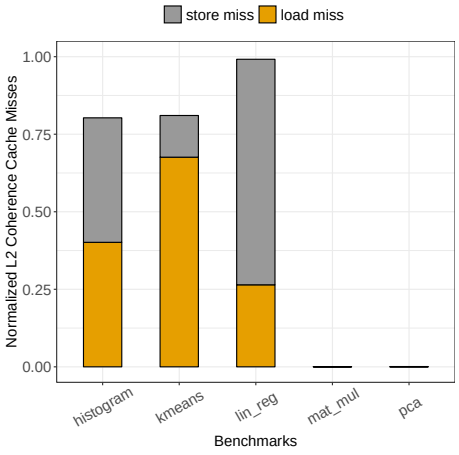
Fig. 3



(a) Fraction of cache accesses that resulted in misses on the L2.



(b) Breakdown of all L2 cache misses into misses on loads or stores.



(c) Breakdown of the L2 cache misses into load misses or store misses due to coherence state. Misses that are not shown are due to capcity and conflict misses.

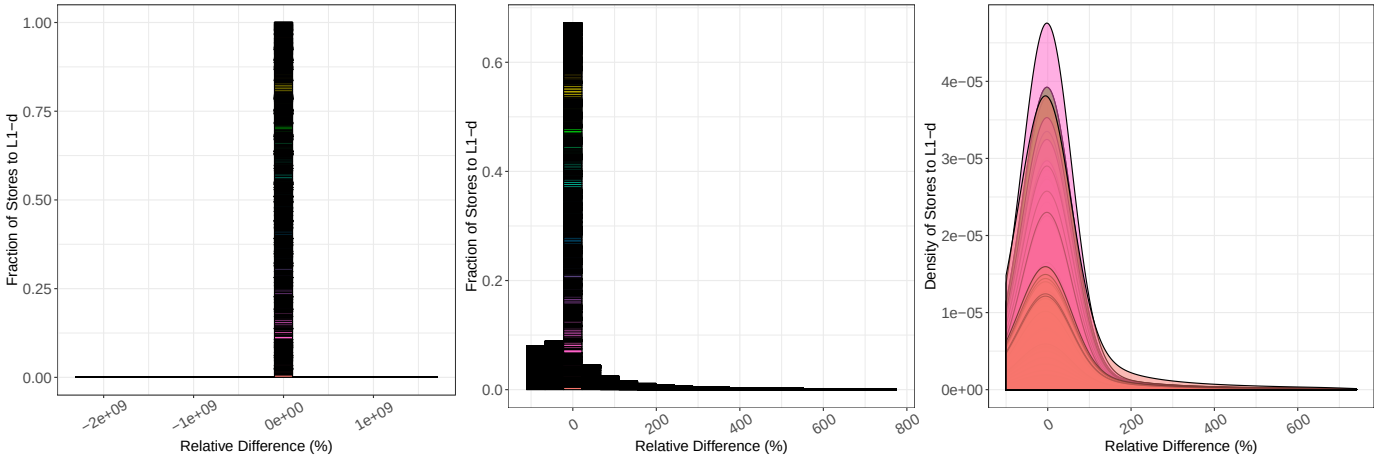
Fig. 4

TABLE II: Store-value similarity for all approximatable stores.

Benchmark	# Stores Sampled	# Unique Addresses	Median	Mean	Std. Dev.	Max	Min
Histogram	1.04×10^9	5142	241	1162.15	5.90×10^6	1.68×10^9	-2.1×10^9
KMeans	193×10^6	1440	7	-449×10^6	904×10^6	2.1×10^9	-2.1×10^9
Linear Regression	773×10^6	40	-2	240.89	793×10^3	2.1×10^9	-2.1×10^9
PCA	502×10^6	428290	-99	-3.91×10^6	92.1×10^6	1.7×10^9	-2.1×10^9
Matrix Multiply	159×10^6	8	78	-3.30×10^6	84.9×10^6	1.68×10^9	-2.1×10^9

TABLE III: Store-value similarity for top 5 traffic generating approximatable stores.

Benchmark	# Stores Sampled	# Unique Addresses	Median	Mean	Std. Dev.	Max	Min
Histogram	35.4×10^5	5	-1	483.07	1.09×10^6	1.68×10^9	-2.1×10^9
KMeans	3.75×10^6	5	0	-320×10^6	790×10^6	2.1×10^9	-2.1×10^9
Linear Regression	136×10^6	5	-1	156.92	752×10^3	2.1×10^9	-2.1×10^9
PCA	313×10^6	5	-1	-2.19×10^6	68.7×10^6	1.7×10^9	-2.1×10^9
Matrix Multiply	113×10^6	5	20	-2.92×10^6	80.0×10^6	1.68×10^9	-2.1×10^9

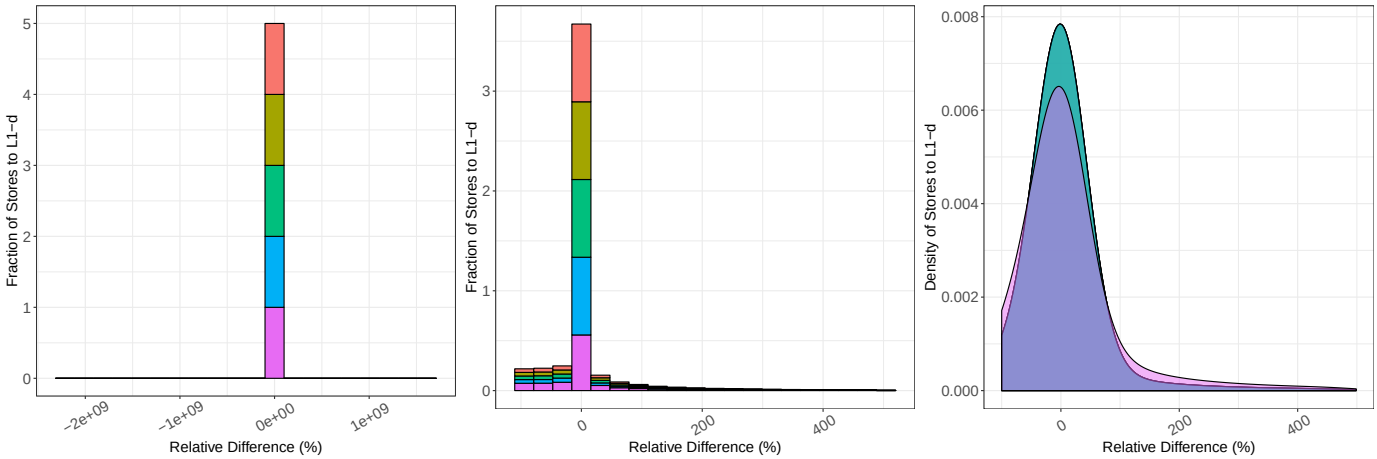


(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 5: Histogram

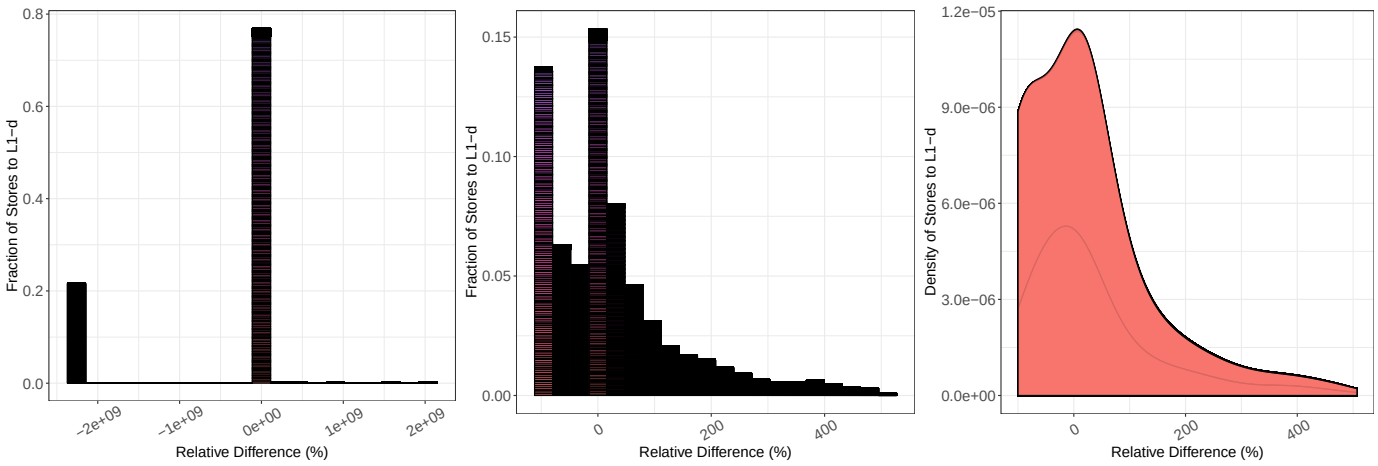


(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 6: Histogram top 5 stores

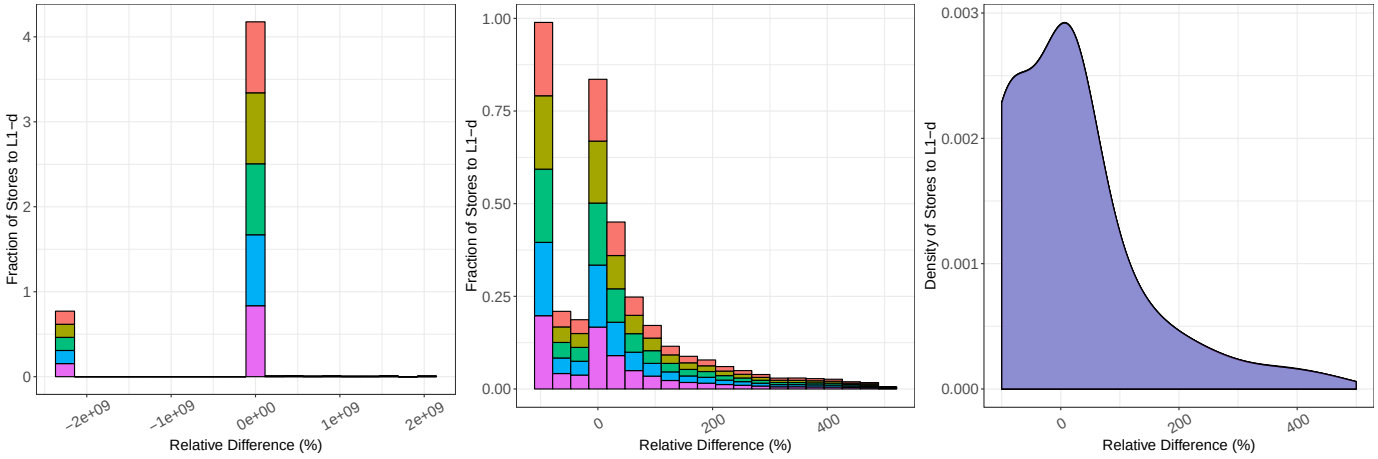


(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 7: Kmeans



(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 8: KMeans top 5 stores

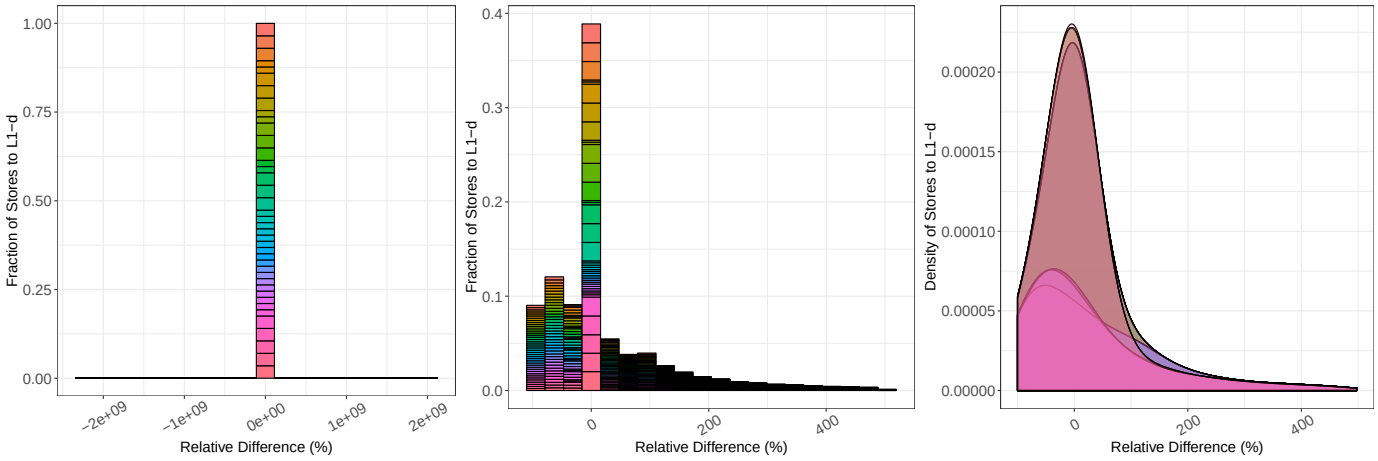
TABLE IV: Stores saved for different level of approximations for benchmark Histogram

Relative Difference (%)	Stores Saved (%)
1	58.02
5	68.62
10	71.16
20	74.33
30	76.97
40	79.40
50	81.78

TABLE V: Stores saved for different level of approximations for benchmark KMeans

Relative Difference (%)	Stores Saved (%)
1	4.86
5	9.46
10	13.14
20	18.70
30	22.94
40	26.51
50	30.25

[6] Tongping Liu and Emery D. Berger. “SHERIFF: Precise Detection and Automatic Mitigation of False Sharing”. In: *Proceedings of the 2011 ACM International*

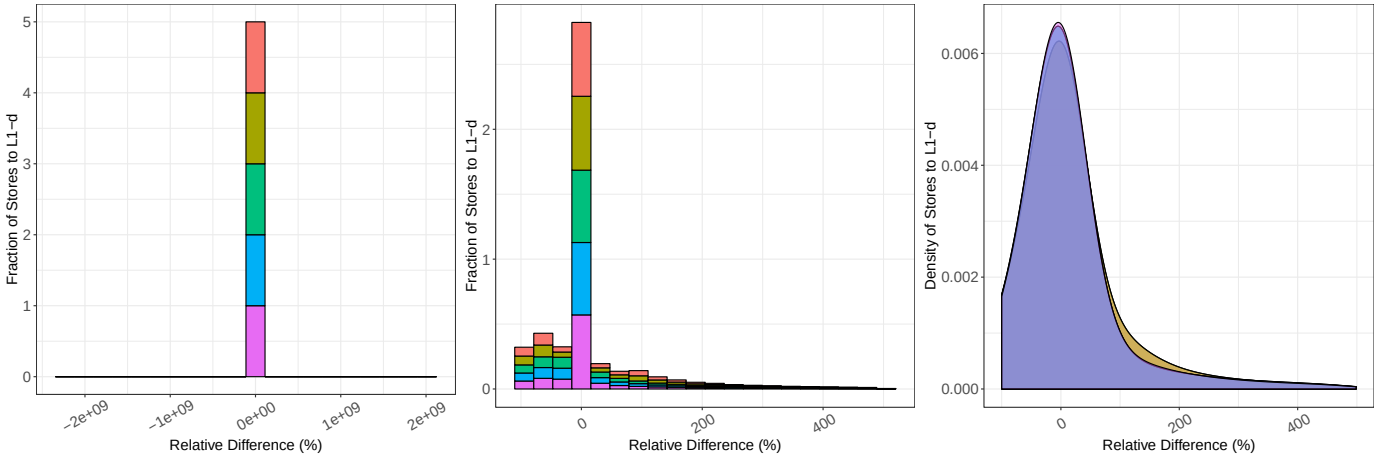


(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 9: Linear Regression



(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 10: Linear Regression top 5 stores

TABLE VI: Stores saved for different level of approximations for benchmark Linear Regression

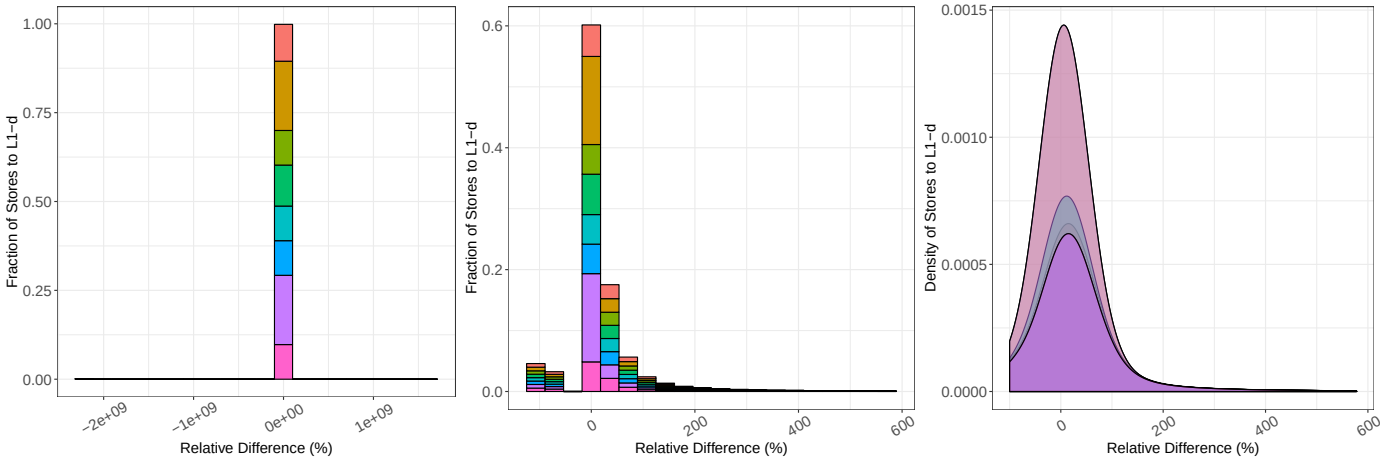
Relative Difference (%)	Stores Saved (%)
1	50.50
5	51.85
10	54.27
20	57.92
30	61.21
40	64.36
50	67.49

TABLE VII: Stores saved for different level of approximations for benchmark Matrix Multiply

Relative Difference (%)	Stores Saved (%)
1	36.68
5	48.46
10	56.32
20	66.09
30	72.12
40	76.28
50	79.25

Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA '11. 2011.

[7] Tongping Liu et al. "PREDATOR: Predictive False Sharing Detection". In: *Proceedings of the 19th ACM*

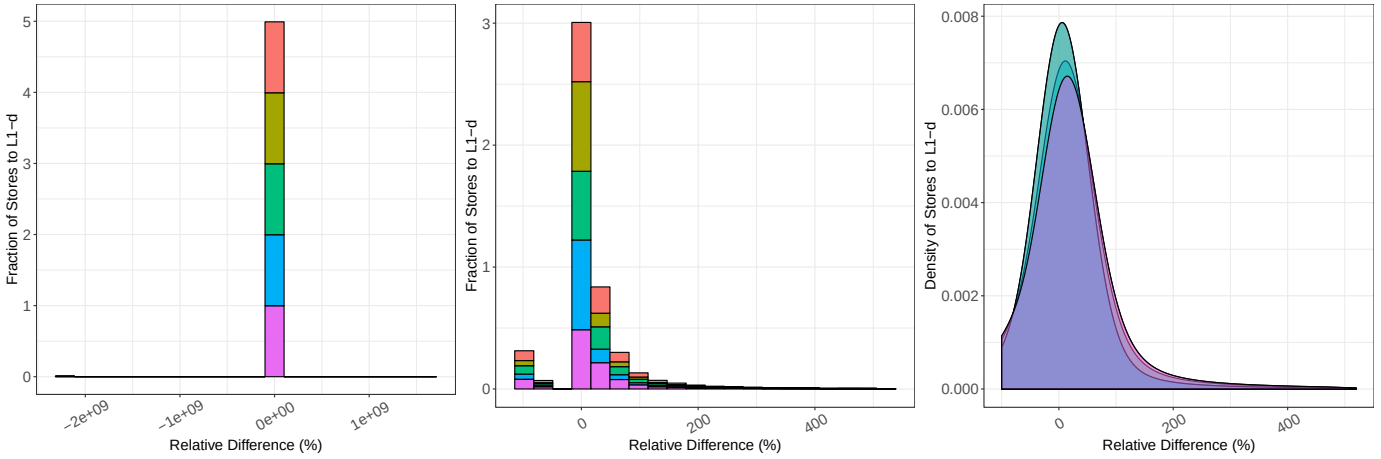


(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 11: Matrix Multiply



(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 12: Matrix Multiply top 5 stores

TABLE VIII: Stores saved for different level of approximations for benchmark PCA

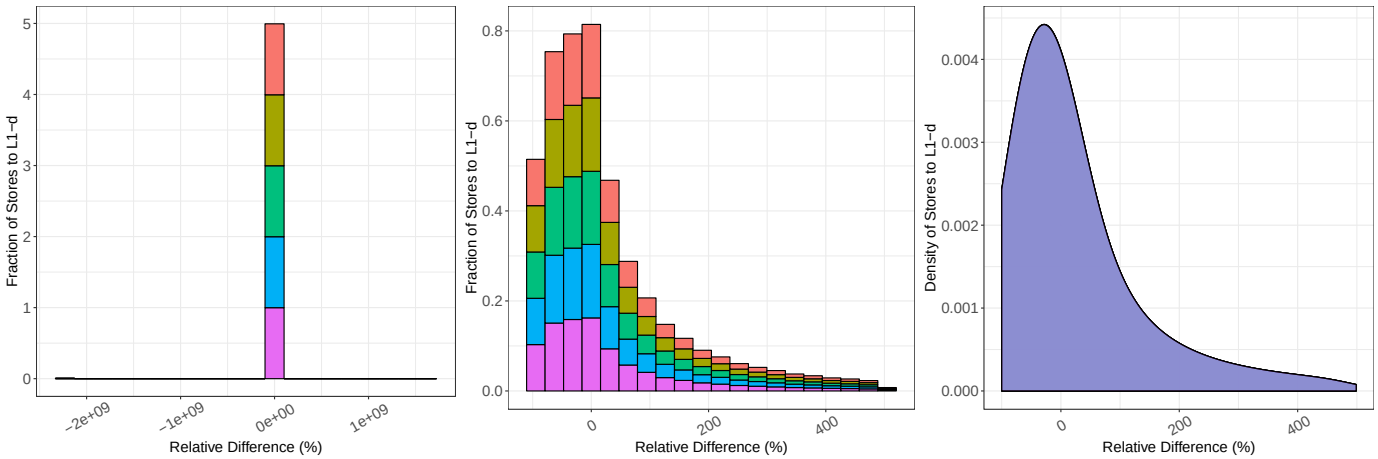
Relative Difference (%)	Stores Saved (%)
1	3.00
5	6.22
10	10.91
20	19.78
30	28.02
40	35.69
50	42.93

SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP '14. 2014.

[8] C. Ranger et al. "Evaluating MapReduce for Multi-core and Multiprocessor Systems". In: *International Symposium on High Performance Computer Architecture (HPCA)*. 2007.

[9] P. V. Rengasamy et al. "Exploiting Staleness for Approximating Loads on CMPs". In: *Parallel Architecture and Compilation (PACT)*. 2015.

[10] X. Wang et al. "A Quantitative Study of the On-Chip Network and Memory Hierarchy Design for Many-Core Processor". In: *International Conference on Parallel and Distributed Systems (ICPADS)*. 2008.



(a) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is the full realtive difference range.

(b) Histogram of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

(c) Density of the fraction of approximateable stores within a store-value relative difference. Shown is focused view between relative percent difference of -500% and 500%.

Fig. 13: PCA top 5 stores