Ghostwriter: A Cache Coherence Protocol for Error-Tolerant Applications

Henry Kao*
University of Toronto
Toronto ON, Canada
h.kao@mail.utoronto.ca

Joshua San Miguel
University of Wisconsin–Madison
Madison WI, USA
jsanmiguel@wisc.edu

Natalie Enright Jerger
University of Toronto
Toronto ON, Canada
enright@ece.utoronto.ca

Abstract

Coherence induced cache misses are an important aspect limiting the scalability of shared memory parallel programs. Many coherence misses are avoidable, namely misses due to false sharing – when different threads write to different memory addresses that are contained within the same cache block causing unnecessary invalidations. Past work has proposed numerous ways to mitigate false sharing from coherence protocols optimized for certain sharing patterns, to software tools for false-sharing detection and repair. Our work leverages approximate computing and store value similarity in error-tolerant multi-threaded applications. We introduce a novel cache coherence protocol which implements an approximate store instruction and coherence states to allow some limited incoherence within approximatable shared data to mitigate both coherence misses and coherence traffic for various sharing patterns. For applications from the Phoenix and AxBench suites, we see dynamic energy improvements within the NoC and memory hierarchy of up to 50.1% and speedup of up to 37.3% with low output error for approximate applications that exhibit false sharing.

Keywords

Cache Coherence, Approximate Computing, Parallel Programming

ACM Reference Format:
Henry Kao, Joshua San Miguel, and Natalie Enright Jerger. 2021. Ghostwriter: A Cache Coherence Protocol for Error-Tolerant Applications. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3458744.3474045

1 Introduction

Proliferation of multi- and many-core processors has made parallel shared-memory applications increasingly important for higher throughput and energy efficiency. Most recent advances focus on improving compute rather than data movement even though energy spent on data movement is significantly greater [7, 20]. Consequently, the communication infrastructure and bandwidth between cores is one of the main bottlenecks [35]. A key component of the communication substrate is the cache coherence protocol which maintains correctness of shared memory among private caches. Although the coherence protocol ensures a system-wide agreement on cache block values, it can lead to communication inefficiencies when multiple writers access data that falls within the same block.

Most cache-coherent chip multiprocessors (CMPs) use write-invalidate coherence protocols; a write to a local cache block invalidates all copies of the cache block within other private caches through invalidation requests. Invalidations operate at a cache block granularity. Coarse granularity is desirable to reduce metadata such as tag storage but can become a bottleneck when multiple cores access the same block repeatedly–as in the case of false sharing. One core's write to a single byte or word in a block invalidates all values within that cache block for all other cores, even if those values are untouched. Subsequent loads and stores to the invalidated blocks, which may still hold values that are technically coherent, result in coherence misses and require the blocks to be fetched from remote caches. Shared data structures are susceptible to superfluous false-sharing coherence misses if they are not programmed skillfully [32]. These misses not only incur additional energy and latency, they also produce avoidable coherence traffic.

Within the last decade, prior works have proposed numerous coherence optimizations in the traditional domain of architecture research [13, 21, 52]. However, our work explores coherence through an emerging paradigm for processor design–approximate computing. As many application domains are inherently error-tolerant (e.g., machine learning, multimedia, scientific computing), we can leverage the approximate computing domain to trade-off computation accuracy for greater energy efficiency and performance. Most exploration of hardware support for approximation targets compute resources [19, 55, 56]. Approximation in the memory system has seen less work, and the designs that exist mostly address memory storage by leveraging value similarity for greater storage capacity and energy efficiency [48–50].

We propose a novel coherence protocol, **`Ghostwriter`**, for error-tolerant applications. **`Ghostwriter`** leverages the inherent value similarity within approximate applications to mitigate the performance degrading effects of false sharing from the source–store instructions. We extend the ISA with a new approximate store instruction used to access shared data structures amendable to approximation. We also implement two new approximate coherence states which allows for approximate stores to update values that show high similarity without broadcasting coherence requests; this reduces both coherence traffic and coherence misses. To summarize, our main contributions are:

*Now at Huawei Technologies Canada.

ICPP Workshops '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-8441-4/21/08…$15.00
https://doi.org/10.1145/3458744.3474045

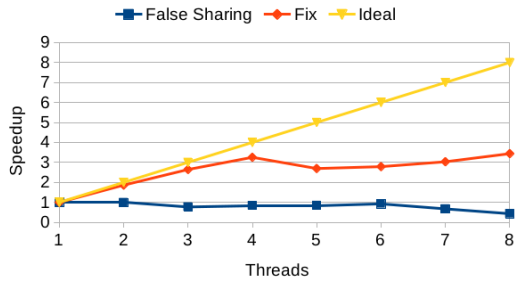


Figure 1: Speedup for the false sharing examples in Listing 1 for different thread counts.

We observe that multi-threaded applications exhibit considerable value similarity between store values and the values being overwritten in a cache block.
We propose **`Ghostwriter`**, a coherence protocol which exploits value similarity to mitigate coherence misses in approximate applications.
We show that our **`Ghostwriter`** protocol provides energy savings of up to 50.1% within the NoC and memory hierarchy (11.2% on average), as well as speedups of up to 37.3% (6.5% on average) for error-tolerant applications that exhibit false sharing, all with very low error.

2 Background & Motivation

False Sharing: In contrast to true sharing (multiple nodes writing to the same address) which is a property of the application, false sharing occurs due to cache coherence and memory accesses occurring at the cache block granularity and is mostly avoidable. However, it is notoriously difficult to manually debug false sharing as it is only implicitly defined in the source code [32]. Implementation of shared data structures can lead to extraneous amounts of cache contention if not programmed carefully, limiting the performance benefits offered by parallelization (up to an order of magnitude [9]). These effects can be found in a range of multi-threaded software from benchmark suites like Phoenix [38] to the Linux Kernel [11].

Listing 1 and 2 illustrates false sharing examples using two different implementations of parallel dot-product. The first is a naive implementation where separate threads directly store their own dot product results into a portion of the shared array `total`. Although the code is simple, it causes substantial cache contention (in the form of coherence misses) through false sharing. Different threads are accessing elements within the shared array `total` which map to the same cache block. The coherence protocol invalidates all shared/exclusive copies of the block for every write to `total`. Increasing thread counts lead to greater amounts of invalidations and coherence misses as shown in Fig. 1, slowing down execution time over single-threaded performance. A simple fix shown in Listing 2 to the the naive implementation is for each thread to write to its own private variable, mapped to different cache blocks. This decreases the amount of sharing, and hence invalidations improving multi-threaded performance using more threads as seen in Fig. 1. Such code transformations can be done automatically by the compiler; however, they are difficult and rarely applied to real-world use cases as they require strict static analysis criteria to be met [37, 54].

As code bases become larger and parallelism becomes more pervasive with increased core counts, locating performance bottlenecks due to false sharing will be an increasingly difficult, yet important problem [34].

Listing 1: False sharing prone parallel dot-product

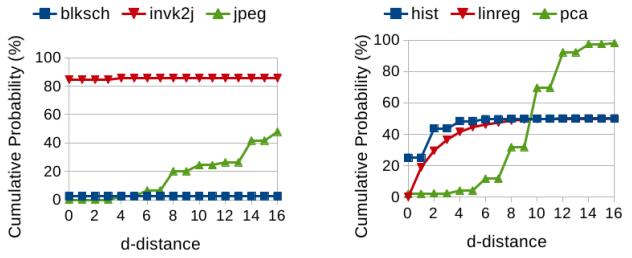
```
int a[N], b[N];
int total[NUM_THREADS];
void dot_product (...) {

    for (i = 0; i < N; i++) { // parallel loop
        total[thread_id] += a[i] * b[i];
    }

}
```

Listing 2: Privatized parrallel dot-product

```
int a[N], b[N];
int total[NUM_THREADS];
void dot_product (...) {
    int sum = 0;
    for (i = 0; i < N; i++) { // parallel loop
        sum += a[i] * b[i];
    }
    total[thread_id] = sum;
}
```

Value Similarity: Prior work leverages the concept of value locality–the recurrence of previously seen values–for architecture optimizations [30, 31]. One form of value locality is bit-wise *value similarity* (or Hamming similarity), which is a measure of how similar values are in binary. For example, values such as 124_{10} (01111100_2) and 127_{10} (01111111_2) can be considered bit-wise similar because only the least significant two bits are different. Values such as 127_{10} (01111111_2) and 128_{10} (10000000_2) are arithmetically close; however, they are not bit-wise similar since all their bits differ. Value similarity has been used for further optimizations, especially in the domain of approximate computing [48–50]. In this paper, we quantify bit-wise value similarity using *d-distance* [57] where the *d* represents the maximum number of least significant bits that differ between two values. Values 121_{10} (1111001_2) and 125_{10} (1111101_2) have 3-distance similarity because the most significant bits are identical up until the 3 least significant bits.

We explore the multi-threaded AxBench [58] and Phoenix [38] benchmark suites for the prevalence of value similarity by comparing memory write values to the value currently in the cache block (irrespective of coherence state). Fig. 2 shows the probability of finding overwritten values with a given d-distance or lower. On average, 22.8% of overwritten values have a 0-distance meaning the incoming value and the value in the cache block are identical in all bits. If the value being overwritten is in a cache block with write permissions, it would be a silent store [27]. Since silent stores have no effect on system state, they add unnecessary memory traffic if the block does not have writing permissions incurring both energy and latency overheads–a problem targeted in prior work [28, 29]. The inherent error-tolerance of approximate applications can allow for some deviation within data values. If we move beyond pure silent stores, we find more opportunity for optimization by using larger d-distances (e.g., 36.4% and 43.7% of written values are 4-distance and 8-distance similar for sampled applications). A possible optimization using bit-wise value similarity is to inhibit a



(a) (b)

Figure 2: Cumulative distribution of d-distance values observed in multi-threaded approximate applications in (a) AxBench and (b) Phoenix.

write to memory if the values are approximately similar; inhibiting writes will yield performance improvements and energy savings. For example, consider an image processing application modifying a 24-bit RGB pixel. Allowing some deviation within the last few bits would alter the blue coloring. It may change the accuracy of the output image; however the change may be imperceptible or acceptable to the user leading to good trade-off for potential energy and performance improvements.

3 Ghostwriter Protocol

We present **Ghostwriter**, a coherence protocol that leverages both error-tolerance and value similarity within approximate applications. **Ghostwriter** extends the baseline protocol with approximate coherence states to allow some data value divergence for select memory addresses to enable approximate execution. Approximatable data structures are labeled with simple programmer annotations that are agnostic of the underlying hardware/coherence protocol [5, 16, 45, 47].

3.1 Programmer Support

Programmers can annotate shared data structures amenable to approximation by specifying the memory locations and d-distance of allowed approximations. The compiler then converts conventional stores to our approximate stores (`scribble` instructions) using prior compiler mechanisms [5] and ISA extensions for approximate programming [16]. The `scribble` instructions are allowed to update values in cache blocks regardless of the coherence state whenever the store value and cache block value differ by less than or equal to the d-distance statically set by the programmer for each approximate region, otherwise falling back to the conventional coherence mechanisms. Compiler static analysis will also pad the annotated heap and stack allocated memory regions to ensure delineation of approximate data and non-approximateable data. A cache block will only contain either approximate data or non-approximate data, not both. Approximatable data structures are usually allocated in long contiguous memory regions; the memory footprint overhead from padding is small. **Ghostwriter** introduces three pragmas for the programmer to annotate the approximate regions of code, and the level of approximation:

`approx_begin(<var>,...)` : Enables approximate (`scribble`) instructions to be generated for the given set of memory locations immediately following this annotation and enables coherence transitions to the approximate states.

`approx_end(<var>,...)` : Disables approximate instructions to be generated for the given set of memory locations following this annotation and disables any further coherence transitions to approximate states.

`approx_dist(<val>)` : Sets the level of approximation for the subsequent approximate instructions where `<val>` is the d-distance. The compiler ensures that the d-distance is legal for the corresponding data type/width to be approximated (i.e., using 8-distance for byte-sized data would allow any value to be written which is an undesirable level of approximation).

Listing 3 shows a simple example of programmer annotations in false sharing prone code of shared arrays `foo` and `bar`. The level of approximation this parallel loop is set using the `approx_dist` pragma specifying the d-distance (4 in this example). The memory locations of the arrays are passed to the `approx_begin` pragma before the approximate computation region to enable `scribbles`. Concurrent approximations of different data structures of different data types/widths are allowed as long as they are annotated by the programmer. Approximation on the data structures can be stopped using the `approx_end` pragma; the blocks are not flushed, and can continue to be used for computation.

Listing 3: Example of approximate programmer annotations

```
int  a[N], b[N];
int    foo[NUM_THREADS];
float  bar[NUM_THREADS];

void approx_kernel (...) {
    #pragma approx_dist(4)
    #pragma approx_begin(foo, bar)
    for (i = 0; i < N; i++) //  parallelizable  loop
        foo[thread_id] += a[i] * b[i];
        bar[thread_id] += a[i] + b[i];
    #pragma approx_end(foo, bar)
}
```

The d-distance settings can be varied throughout the applications in different approximate regions by using the `approx_dist` pragma and can be further fine-tuned via profile-guided optimization (PGO) and/or auto-tuning techniques [5, 14, 19, 44, 46]. Each new d-distance setting would require our modified cache controller (see Sec. 3.4) to be re-programmed. An additional instruction `setaprx` is used to update the controller, which encodes as an immediate value for the new d-distance to be programmed. Instruction `endaprx` is used to denoted the end of the approximate region, and disable approximate writes. The `setaprx`, `endaprx`, and `scribble` instructions are implemented in unused opcodes of the existing ISA. No other hardware changes in the instruction pipeline are needed. Setting new d-distances should be used sparingly (e.g., not in extensive loops); used properly, they will have negligible effect on dynamic instruction count and execution time. Shared data structures that see the most benefit from approximation are ones being frequently updated in long loops or frequent function calls. Note that the programmer does not have to exactly identify sharing within the code, only regions where memory is shared and approximatable. The programmer should also treat the approximate data as a volatile type – the approximate data can change at any

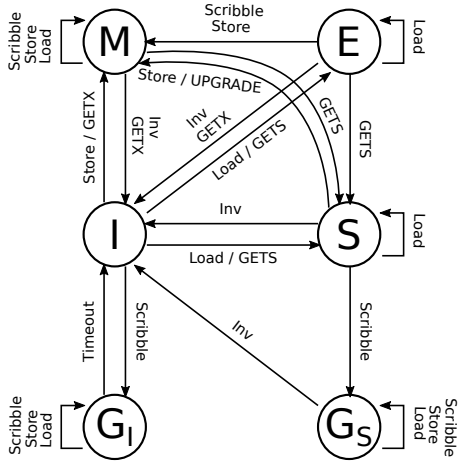


Figure 3: `Ghostwriter` protocol with approximate states G_S and G_I implemented on top of an MESI protocol. Note that state I means the block is present in the cache (i.e., the tag exists) but in the invalid coherence state.

time. If one thread exits the approximate region where another thread is still in the approximate region, these values are still allowed change. Data to approximate should be consistent across threads. In cases where approximation is not consistent across all threads, the conservative approach is to not annotate that data structure for approximation. False and true sharing are mitigated during runtime as explained in the following sections. Structures for control flow and memory addresses should not be annotated as it may cause unexpected execution.

3.2 Protocol Description

In conventional write-invalidate coherence protocols, cache blocks in shared state (**S**) have read-only permissions. These blocks only allow loads to hit as the data contained within them are unmodified and multiple copies may be shared between private caches. Blocks in the invalid state (**I**) contain stale and incoherent values and both loads and stores will miss when accessing invalid blocks. A cache's block is invalidated when some other cache wishes to update to its own copy of that block, marking all other copies that exist as incoherent and thus invalid. Blocks in **S** state and **I** state share a commonality–both send out coherence requests to obtain exclusive access to the block upon a store. This results in the invalidation of all other copies of the block held within every other private cache. Although the write-invalidate procedure maintains a system-wide agreement on coherent cache block values, it can lead to performance degradation for certain sharing patterns. More specifically when stores from different cores continuously access the same block, this causes a ping-pong effect in invalidations and hence an increase in coherence misses, incurring both energy and latency penalties.

`Ghostwriter` introduces two new coherence states to support approximate execution of error-tolerant applications, G_S and G_I. The intuition behind these states are to allow private caches have their own *locally modified* (hidden from the system-wide view) copies of shared cache blocks; these states are entered when `scribbles` wish to update data that is approximately similar to what is already present. These approximate states mitigate the detrimental effects of stores on blocks in **S** and **I** state by suppressing requests for exclusive access on approximatable stores (`scribbles`). These stages can be added to most existing protocols. Without loss of generality, we use a a baseline MESI write-invalidate directory protocol for **`Ghostwriter`** as seen in Fig. 3.

Approximate state G_S reduces the amount of **UPGRADE** requests and subsequent invalidations for stores to approximatable data. A store to approximatable data that meets current d-distance set by the application enables a `scribble` instruction to transition from shared to G_S. The `scribble` can immediately update the data within the shared cache block with a value that is approximately similar, and the block is now *locally modified*–hidden from global view. Blocks in G_S are granted both read and write permissions, so conventional loads, stores, and `scribbles` all results in cache hits. The directory sharer list still holds the cache as sharer, and transitions to **I** when an invalidate request arrives from another core requesting exclusive access to the block due to the execution of a conventional store.

Approximate state G_I reduces the amount of get exclusive (**GETX**) requests for stores to approximatable data. Conventional stores to a block in the **I** state would forward **GETX** requests to the cache that currently has exclusive access to the block (i.e., in the modified (**M**) or exclusive (**E**) state). The block data is transferred to the requestor and invalidated in the responder. A `scribble` to a block in **I** transitions to G_I without sending a **GETX**. The data within the invalid block is also updated immediately with a value that is approximately similar. Similar to blocks in G_S, blocks in G_I are now locally modified and subsequent loads, stores and `scribbles` have read and write permissions to the block. To prevent unlimited read and writes, blocks in G_I transition back to **I** using a periodic timeout for each cache controller.[1] The timeout is necessary since the transition from **I** to G_I does not add the cache as a sharer in the directory, so no invalidation requests would reach it as opposed to the case of G_S. Adding G_I blocks as sharers would require changes further into the cache hierarchy/directories which can be a substantial implementation and verification task, especially for more complicated protocols [2]. We keep our modifications to the protocol simple and local to the L1 level of the hierarchy.

3.3 Protocol Operation

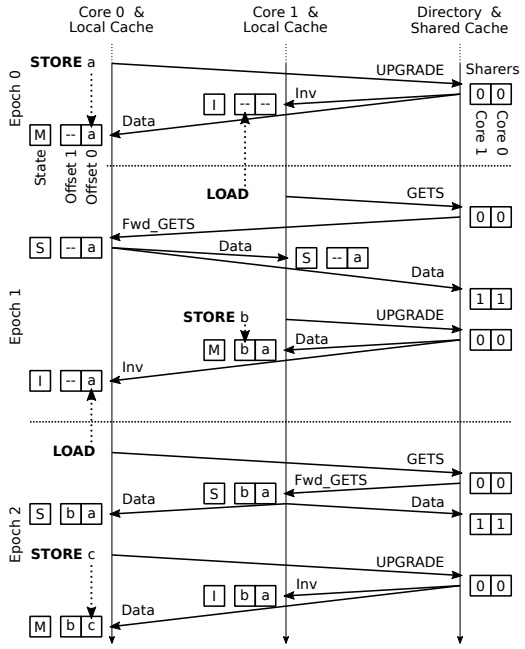
We illustrate the operation of approximate states G_S and G_I in the **`Ghostwriter`** protocol through two example sharing patterns. Time is artificially delineated using epochs for the sake of clarity.

Migratory sharing is when different threads first load, then store to the same cache block–the block migrates between different threads. An example is depicted in Fig. 4a for our baseline MESI protocol. The sharing pattern is as follows:

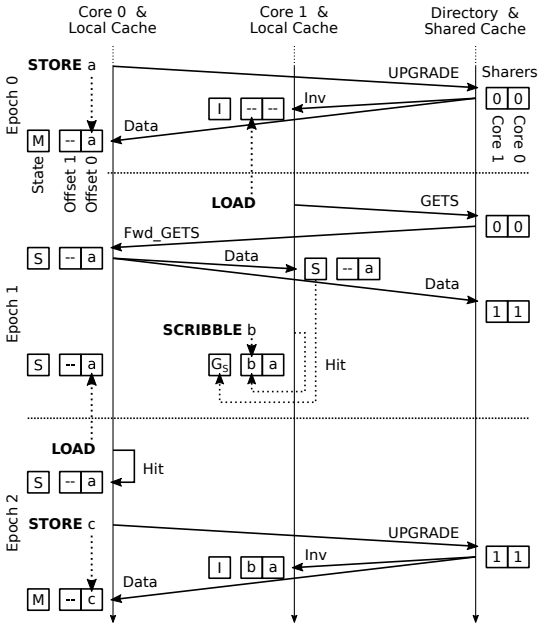
In Epoch 0, Core 0 stores value $< a >$ to a shared block at offset 0 resulting in an **UPGRADE** request being sent to the directory. All shared copies are invalidated so that Core 0 has exclusive access and completes the write.

In Epoch 1, Core 1 attempts to read from block offset 1, however misses due to the previous invalidation. A **GETS** request is issued to obtain read permissions (**S** state). The following store of value $< b >$ to block offset 1 misses due the the block being in **S**. An

[1] We empirically set this value to 1024 as detailed in Sec. 4.4.



(a) Migratory false-sharing pattern in baseline MESI directory protocol. Core 0 and Core 1 each first loads, and then stores values to block offsets 0 and 1, respectively.



(b) Migratory false-sharing pattern using `Ghostwriter` approximate state G_S. Coherence transactions are reduced by allowing the same block in Core 0 and Core 1 to be valid, but have incoherent data.

Figure 4: Migratory sharing within (a) baseline protocol and (b) `Ghostwriter` protocol.

UPGRADE request is sent, invalidating all shared copies before obtaining exclusive access block and writing the value.
In Epoch 2, Core 0's load will miss due to the invalidation from Epoch 1, and the process repeats itself.

Note that this migratory sharing example also exhibits false sharing since Core 0 and Core 1 are loading and storing to different offsets in the same cache block. Even though both cores access different memory addresses, this false sharing generates avoidable coherence transactions and subsequently, unnecessary coherence misses. **`Ghostwriter`** is able to minimize these negative effects using approximate state $\mathbf{G_S}$ illustrated in Fig. 4b as follows:

In Epoch 0, the store follows the same behaviour as the store in the baseline MESI example.
In Epoch 1, the initial load also proceeds the same ways as in the baseline. However, if instead the store from Core 1 is a `scribble`, the block transitions from state **S** to approximate state $\mathbf{G_S}$ and the value is updated immediately without needing exclusive access.
In Epoch 2, the load from Core 0 hits since the block was not invalidated from Core 1's `scribble` in the Epoch 1, hiding the coherence misses seen in the baseline example.

Note that in this example, both Core 0 and Core 1's loads still read correct values since they are writing the different block offsets. If Core 0's load in Epoch 2 were to read from offset 1, a stale would be returned leading to approximate execution.

Producer-consumer sharing exists when one producer thread writes a value followed by one or more consumers threads reading that value. In addition, the producer is not fixed to a single node–more often the producer changes between nodes [6]. Fig. 5

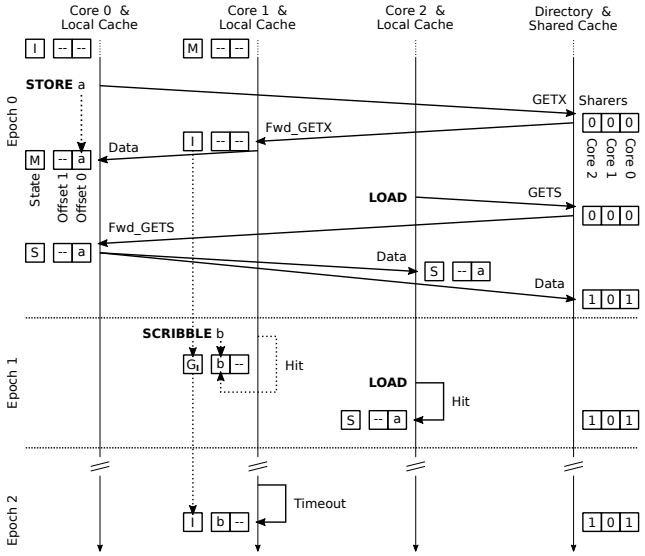


Figure 5: Producer-consumer pattern using `Ghostwriter` approximate state G_I. Core 0 is the first producer storing to an invalid block in offset 0, Core 1 is the next producer. Coherence transactions are reduced by allowing `scribbles` to update invalid blocks.

illustrates an example of producer-consumer sharing in **`Ghostwriter`** using the $\mathbf{G_I}$ approximate state as follows:

In Epoch 0, assume Core 1 initially holds the block in **M** state. Core 0 acts as the producer storing value $< a >$ at offset 0 by first requesting exclusive access through a **GETX** request. The value

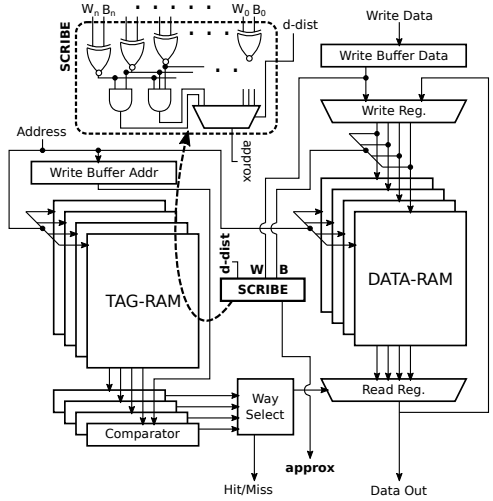


Figure 6: Cache controller modifications to support the `scribble` ISA extension. An additional module, `scribe`, is needed to compare incoming write data with block data.

is updated once Core 1 forwards the block data to Core 0. Core 2 acts as a consumer and sends a **GETS** to the directory for shared access. The load completes once data is forwarded from Core 0. Both Core 0 and Core 2 share the block at the end of Epoch 0, with Core 1's block invalidated. This depicts producer-consumer sharing in the baseline MESI protocol.

In Epoch 1, Core 1 is the producer seeking to update block offset 1 with value $< b >$. Conventionally this would repeat the producer-consumer process in Epoch 0, however if Core 1's store is approximatable (`scribble`), the write is immediate and transitions to approximate state $\mathbf{G_I}$. Following loads, stores and `scribble` instructions from Core 1 hit in its cache. Core 2 acts as consumer again, however since no **GETX** requests were sent by Core 1, the load hits reducing both latency and coherence traffic.

After some time in Epoch 2, the timeout of Core 1's cache controller transitions the block back to the **I** state from $\mathbf{G_I}$ to return to coherency, but loses the update from the `scribble` instruction.

Note that if Core 2's load in Epoch 1 accesses block offset 0, it reads the correct value. However if Core 2 reads from offset 1, it will execute on a stale value – approximate execution. **`Ghostwriter`** not only minimizes the impacts of coherence misses due to false sharing in both examples, it also hides coherence misses due to true sharing by sacrificing accuracy in shared approximatable data.

3.4 Cache Controller

Hardware support in the cache controller for the `scribble` ISA extensions is shown in Fig. 6 which includes an additional module, `scribe`, made of XNOR equality comparators. The `scribe` module takes write data (W) and the cache block word (B) and compares them given the d-distance defined by the programmer for the application. Our design closely follows existing cache controller designs [36]; upon a store, the whole block is first read and registered into the Write Reg. The corresponding word is updated before the whole block is written back into the data RAM. Write data and the block data are compared during the write procedure. The `approx` signal is set if `scribe` determines both values meet the set d-distance and `scribble` coherence state transitions are enabled as in Fig. 3, otherwise it falls back to the baseline protocol state transitions. A minimum of one cycle latency is needed to determine a tag hit before writing to the block in which the `scribe` can execute in parallel, thus it is not on the critical path [1].

Signed integer values -1 (0xFFFF) and 0 (0x0000) can be considered arithmetically close, but are completely different bit-wise using our definition of d-distance. Small d-distances only apply to the mantissa in floating point values which can limit possible approximation opportunities. These issues along with supporting custom data types can be solved at the expense of greater hardware complexity which is left as future work.

3.5 Error Tuning and Bounding

`Ghostwriter` aims to open a new dimension of optimizations for current and future error-tolerant applications. Our technique can enable both developers and end-users to exploit the features in approximate coherence to obtain better energy-efficiency and performance, however, there are some limitations in which `Scribbles` can introduce errors to approximate data:

Invalidation of both $\mathbf{G_S}$ (from invalidations requests) and $\mathbf{G_I}$ (from periodic timeouts) returns the blocks to system-wide coherency again, however updates made to the data within $\mathbf{G_S}$ and $\mathbf{G_I}$ will be lost upon returning to **I**.

When two or more cores hold a block in either approximate states, neither core will see the updates made by any other core to their copy of the block in $\mathbf{G_S}$ or $\mathbf{G_I}$.

When a block in either approximate state is evicted from the cache (e.g., from replacement request), the data updates made are forfeited.

During context switches and thread migrations, The approximate data cannot be switched/migrated since approximate blocks are not tracked by the directory; the data updates are forfeited.

A difficult challenge that approximate computing techniques face is ensuring output error is bounded. Even though an approximate execution of an application may yield low error values on average, the worst-cases errors can be high [24]. A pathological case in **`Ghostwriter`** occurs when multiple cores are accessing copies of a cache block in approximate states $\mathbf{G_S}$ and/or $\mathbf{G_I}$, and one core is continuously updating a value that meets the d-distance setting for approximation. In this rare case, the values seen by different cores can be considerably different and approximation can be unbounded. Prior work in approximate computing has explored schemes that can detect and resolve unbounded approximation to an acceptable error level. These techniques span the stack from approximate programming languages and static analysis that expose the recovery mechanisms to the programmer [3, 10], to light-weight dynamic schemes that monitor error during runtime [17, 24, 25, 43]. **`Ghostwriter`** also allows for fine grained tuning of an application by the programmer using d-distance, however it may be a hassle if details of the application are not known. In addition to PGO techniques [14, 44], we can also employ existing approximate auto-tuning frameworks to automatically select the approximate regions and d-distance for an output quality target specified by the user [5, 19, 46] Work on error control/recovery mechanisms, PGO, and auto-tuning is orthogonal to ours, but can be readily applied

Table 1: Simulation Configuration

Parameter	Values
Cores	24 in-order cores, X86 ISA, 1GHz
OS	Ubuntu 16.04
L1	Private 32kB D-Cache/32kB I-Cache, 2-Way Set Assoc., 64B Block, Pseudo-LRU, 2-cycle
L2	Shared, 128kB per core, 8-Way Set Assoc., 64B Block, Pseudo-LRU, 10-cycle
Coherence	**Ghostwriter** Protocol (Baseline MESI), d-distance 4 and 8, 1024-cycle $\mathbf{G_I}$ Timeout.
Network	Mesh, XY Routing, 1-cycle router, 1-cycle link, 4 Directory Controllers at Mesh Corners
DRAM	2GB, DDR3 1600MHz

Table 2: Benchmarks

Application	Domain	Input	Error
Phoenix			
histogram	Image Processing	400MB image	MPE
linear_regression	Machine Learning	50MB file	MPE
pca	Machine Learning	4MB matrix	NRMSE
AxBench			
blackscholes	Financial Analysis	200K options	MPE
inversek2j	Robotics	1000K points	NRMSE
jpeg	Image Compression	512x512 RGB	NRMSE

to **Ghostwriter** to ensure worst-case error is bounded for better reliability.

3.6 Memory Consistency and Correctness

`Scribbles` from different cores may access the same approximate datum simultaneously during data races, and the value of a read from an approximate state may not yield up-to-date values. Hence **Ghostwriter**'s approximate state may not follow the consistency model of the underlying hardware. We relax the memory model for approximate data as with prior work in this domain for improved energy efficiency and performance [41, 46, 50]. However **Ghostwriter** still maintains strict consistency for non-approximate data since its follow the baseline coherence protocol. Only data that is labeled as approximate may not follow the consistency model of the underlying hardware, however this is practical since we are operating on error-tolerant applications where imprecise execution of approximate data is accepted.

4 Evaluation

We implement **Ghostwriter** on top of a MESI write-invalidate directory protocol and evaluate in full-system gem5 [8]. The simulated CMP consists of 24 cores, each with private L1 caches and a slice of physically distributed, logically shared L2 cache. Timeout for approximate state $\mathbf{G_I}$ is set to 1024 cycles. Table 1 lists detailed simulation parameters. We model cache and DRAM energy using CACTI [33] and NoC energy using DSENT [53]. Table 2 lists the applications we use from various error-tolerant domains from Phoenix [38] (multi-threaded using pthreads), and AxBench [58], which we multi-thread ourselves using OpenMP. Output error is either measured in maximum percent error (MPE) or normalized root-mean-squared error (NRMSE) depending on suitability for each application [4]. Although **Ghostwriter** supports different d-distance setting within each application, we fix it to 4-distance or 8-distance for all approximatable regions of code for simplicity.

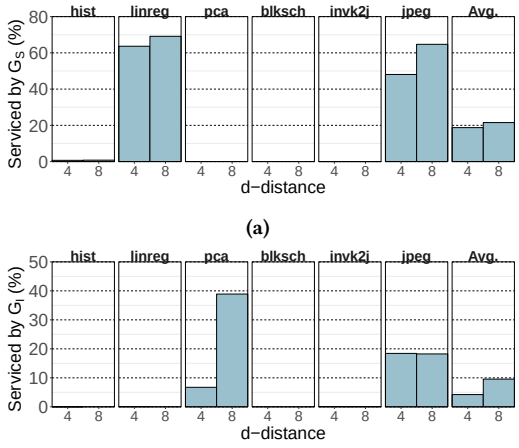


Figure 7: Percentage of stores that would have missed on: (a) S but were serviced by $\mathbf{G_S}$, (b) I but were serviced by $\mathbf{G_I}$.

4.1 Approximate States Utilization

We first look at utilization of **Ghostwriter**'s approximate states $\mathbf{G_S}$ and $\mathbf{G_I}$ using 4-distance and 8-distance. Fig. 7a shows the percentage of stores that would have missed on read-only blocks (**S**) that are serviced by $\mathbf{G_S}$ through the `scribble` ISA extension. For `linear_regression`, sees 63.7% to 69.1% of stores to **S** that are serviced by $\mathbf{G_S}$with 4 and 8-distance. Setting larger d-distance values for each application allows for more difference in the least significant bits between the new store value and the value being overwritten, hence more `scribble` state transitions to $\mathbf{G_S}$. Similarly, benchmark `jpeg` observes an increase in store/`scribble` hits on $\mathbf{G_S}$ from 48% to 64.7%. On average, approximate state $\mathbf{G_S}$ services 18.7% and 21.5% of store coherence misses on **S** for d-distance of 4 and 8 respectively.

Fig. 7b shows the percentage of stores that would have missed on invalid blocks but now hit due to **Ghostwriter**'s $\mathbf{G_I}$ state. Comparable to $\mathbf{G_S}$, increasing d-distance with the application from 4 to 8 allows for more `scribbles` to be enabled. In the case of `pca`, $\mathbf{G_I}$ services 3.7% to 38.9% of the stores that would have missed on invalid blocks in the baseline protocol. The large jump in $\mathbf{G_I}$ utilization comes from only 4.1% of updating values having 4-distance in contrast to 31.8% for 8-distance for the `pca` benchmark. On average for 4-distance and 8-distance, $\mathbf{G_I}$ reduces store coherence misses on **I** by 4.2% to 9.7%, respectively.

4.2 Coherence Transaction Reduction

Approximate states $\mathbf{G_S}$ and $\mathbf{G_I}$ can hide a considerable percentage of store misses on shared and invalid blocks as seen in Fig. 7. However, an application may not see much benefit from **Ghostwriter** if coherence misses only make up a small percentage of total misses. Reduction in coherence transactions can give insight to how much the approximate states affect the overall application. Fig. 8 shows the reduction in the L1D cache coherence requests, namely **GETX**, **UPGRADE**, **GETS**, and data transfers.

`Linear_regression` observes significant false sharing, and hence significant coherence misses on its `lreg_args` data structure. Over 12% of all stores in `linear_regression` miss on shared blocks, and

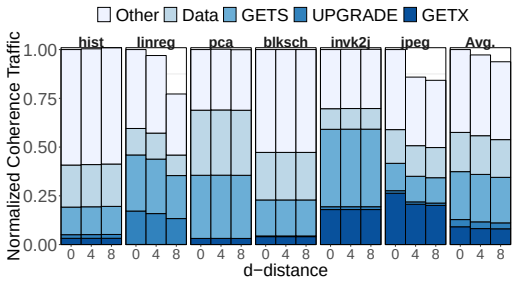


Figure 8: Coherence traffic reduction for `Ghostwriter` using d-distance 4 and 8, with 0 being the baseline MESI protocol.

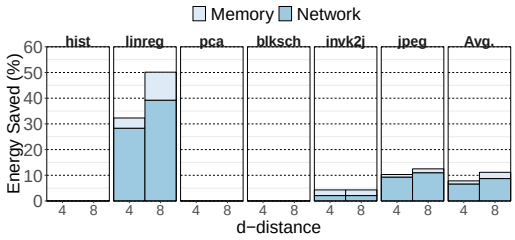


Figure 9: NoC and memory hierarchy dynamic energy savings for `Ghostwriter` using 4-distance and 8-distance.

9% of all loads miss on invalid blocks. Each thread is passed an `lreg_args` via pointer, however the size of the structure is 52B (smaller than a typical 64B block) resulting in multiple structures mapping to the same block. Different threads then read and update their own memory addresses that are mapped to the same blocks, exhibiting migratory false sharing. There is a 22.8% reduction in coherence transactions using 8-distance since 69.1% of the stores that would have missed, instead hit on blocks in $\mathbf{G_S}$. It comes as a direct result of reducing 22.5% of **UPGRADE** requests that would have been sent by conventional stores to blocks in **S**. `jpeg` exhibits a mixture of migratory sharing and producer-consumer sharing spread across multiple shared data structures, using both $\mathbf{G_S}$ and $\mathbf{G_I}$ approximate states. Conventional stores to invalid cache blocks would send **GETX** requests to the directory. However, transitions to $\mathbf{G_I}$ using 8-distance `scribbles` directly reduce the amount of **GETX** requests by 23.6% for an overall reduction of 15.8%. On average across the sampled applications, we see an reduction in coherence transactions of 2.75% using 4-distance and 6.25% using 8-distance.

`pca` has a small percentage of coherence misses, 0.1% out of all cache accesses compared to 10.1% for `linear_regression`. Even though 38.8% of stores to invalid blocks in `pca` are serviced by the $\mathbf{G_I}$ state, the small percentage of coherence misses make **`Ghostwriter`**'s impact inconsequential. Both `histogram` and `blackscholes` show similar behaviour with negligible amount of coherence misses (0.2% and 0.3%, respectively). Prior software tools have found false sharing within `histogram`'s shared array `arg.blue` [12]; however, false sharing is dependent on allocation and scheduling of threads. Given our machine setup, we observed very little false sharing within `histogram` during runtime. Even if data structures are susceptible to false sharing, **`Ghostwriter`** will have little to no impact on the application's performance if this false sharing does not present itself during execution in the form of coherence misses. If it does, then **`Ghostwriter`** minimizes the impact on-the-fly as in the case for `linear_regression` and `jpeg`.

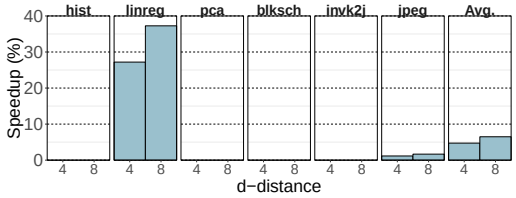


Figure 10: Speedup for `Ghostwriter` using 4-distance and 8-distance.

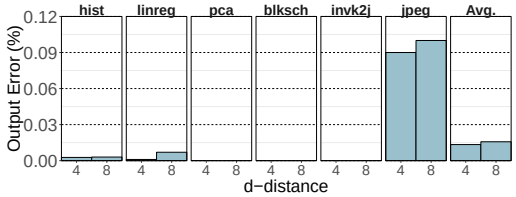


Figure 11: Output error for `Ghostwriter` using 4-distance and 8-distance. Note the scale on the y-axis only goes up to 0.12%

4.3 Energy, Performance and Output Error

Dynamic energy saved using **`Ghostwriter`** for both the NoC and the memory hierarchy normalized to the baseline MESI protocol is shown in Fig. 9. The memory hierarchy includes accesses to L1 and L2 caches, as well as main memory. Improvements seen by using **`Ghostwriter`** is proportional to the amount of false sharing and hence coherence misses within each application during runtime. **`Ghostwriter`** has the largest performance impact on `linear_regression` due to significant coherence misses on its `lreg_args` data structure. The reduction in coherence transactions and consequently network traffic provides up to 32.3% and 50.1% energy savings within the NoC and memory hierarchy for 4-distance and 8-distance. `jpeg` uses both approximate states $\mathbf{G_S}$ and $\mathbf{G_I}$ for dynamic energy improvements up to 10.3% and 12.5% for 4-distance and 8-distance. On average, across the sampled applications **`Ghostwriter`** delivers dynamic energy savings of 7.8% and 11.2% for 4-distance and 8-distance within the NoC and memory hierarchy.

Speedup is shown in Fig. 10. Similar to the dynamic energy reduction, speedup is also proportional the amount of coherence misses **`Ghostwriter`** can mitigate. We see up to 27.2% and 37.3% speedup in applications with extensive false sharing (`linear_regression`), and on average 4.7% to 6.5% speedup across all benchmarks for 4-distance and 8-distance. Note that **`Ghostwriter`** has no negative impact on applications that do not exhibit false sharing; **`Ghostwriter`** provides the same performance as the baseline protocol.

Fig. 11 shows output error using **`Ghostwriter`**. The average percent error across all sampled applications is extremely low, less than 0.02% for both 4-distance and 8-distance demonstrating the applications' resiliency to approximation. In the case of false sharing under **`Ghostwriter`**, different threads can read and update the same cache block without the ping-pong effect of passing the block back and forth as seen in Fig. 4. As long as both threads write to different memory address within the block, they still execute on correct data. Errors are introduced for true sharing when different threads are accessing the same memory address in the same block in which case **`Ghostwriter`** allows threads to update simultaneously without seeing updates from the other threads. Data updates are also lost when $\mathbf{G_I}$ timeouts to **I**, or $\mathbf{G_S}$ gets invalidated to **I**;

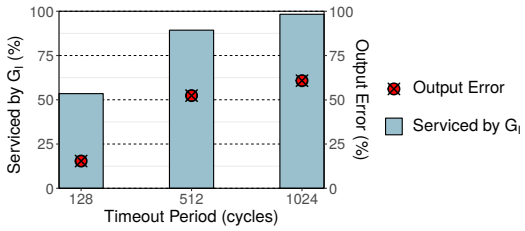


Figure 12: Utilization of G_I state (bars) and output error (points) sensitivity to different timeout periods for false sharing microbenchmark using 4-distance.

however, the inherent value similarity and limiting d-distance to 4 and 8 within the applications palliates the shared data divergence upon `scribble` updates.

`Ghostwriter` does not provide performance gains nor does it degrade performance for applications that do not show false sharing during execution time. It also does not introduce error to these applications. At worst, the application performs as it would in the baseline protocol as seen in benchmarks `histogram`, `pca` and `blackscholes`.

4.4 Timeout Sensitivity

Fig. 12 shows a sensitivity study for the timeout period used to transition $\mathbf{G_I}$ back to invalid. We test the effect of different timeout period on the utilization of $\mathbf{G_I}$, namely the percent of stores that would have missed on **I**, but was serviced by $\mathbf{G_I}$ and output error. The microbenchmark (`bad_dot_product`) from Listing 1 is used to thoroughly exercise the $\mathbf{G_I}$ state. Inputs to the microbenchmark are 8 million integers ranging in values from 0 to 255; we test with 4-distance `scribbles`. Larger timeout periods see greater utilization of the $\mathbf{G_I}$ state, where stores and `scribbles` can update values immediately, reducing coherence transactions, up to 72.4% using timeout period of 1024 cycles. Although longer timeout periods result in more store/`scribble` hits on blocks in $\mathbf{G_I}$, the updates are lost once the timeout period ends so shorter timeout periods lose less updates, and longer periods lose more updates. Error (in MPE) increases from 15.3% using 128-cycle timeout to 60.8% using 1024-cycle timeout; however, this microbenchmark exaggerates the effects of false sharing and is not representative of real applications. We use a 1024-cycle timeout period for $\mathbf{G_I}$ for greater coherence traffic reduction since our applications show high tolerance to error.

5 Related Work

Coherence Optimizations: Numerous cache coherence protocols have been proposed over the past decades to improve performance on specific sharing patterns such as migratory sharing [22, 52], and producer-consumer sharing [13, 23]. Other proposed protocols dynamically adapt to different sharing patterns [21, 39]. Alternative software-hardware codesign approaches exposes coherence to programmers through software primitives, as in user-level/cooperative shared memory [18], Stanford FLASH [26], Wisconsin Tempest/Typhoon [40], CACHET [51], and Denovo [15]. These require knowledge of underlying hardware which can prove too difficult for most programmers [21], and can have high implementation complexity as they propose completely new protocols, and require relaxed consistency models.

Approximate Computing: Prior works in approximate computing has explored value similarity [50]. Doppleganger [49] and Bunker Caches [48] exploit value similarity within cache blocks to improve performance and energy efficiency within memory accesses. Cache coherence has not been well explored within the approximate computing domain. To the best of our knowledge, Rengasamy et al. [42] are the first to propose approximate coherence protocols. Their work stems off of Coherence Decoupling [21], which speculatively executes on stale data from all invalid cache blocks while waiting for coherent data to arrive. However instead of roll-backs when speculative execution in incorrect, they continue computing with the stale values, hence the approximate execution. We extend work in this space by considering approximation to stores that induce false-sharing based cache coherence misses.

6 Conclusion

Multi-threaded shared-memory applications are pervasive as computers progress towards higher core counts. Shared data structures can bottleneck an otherwise performant parallel application due to false-sharing induced coherence cache misses and coherence traffic. As a result, we present a novel cache coherence protocol for the approximate computing domain names **`Ghostwriter`**. **`Ghostwriter`** implements approximate states that reduces the amount of coherence traffic due to stores to shared and invalid cache blocks resulting in fewer coherence misses. We leverage the intrinsic error-tolerance and value similarity present multi-threaded approximate applications to trade off accuracy in shared approximate data for energy and performance improvements. We show that **`Ghostwriter`** protocol can achieve energy savings up to 50.1% within the NoC and memory hierarchy, and speedups of 37.3% for error-tolerant applications that exhibit false sharing, all with very low output error.

References

[1] [n.d.]. ARM1156T2-S Technical Reference Manual. https://developer.arm.com/docs/ddi0338/latest/level-one-memory-system/cache-organization. Accessed: 2019-10-10.

[2] [n.d.]. gem5 MOESI CMP directory. http://gem5.org/MOESI_CMP_directory. Accessed: 2019-03-04.

[3] Sara Achour and Martin C. Rinard. 2015. Approximate Computation with Outlier Detection in Topaz. In *Proc. of the 2015 Int'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*.

[4] Ismail Akturk, Karen Khatamifard, and Ulya R. Karpuzcu. 2015. On Quantification of Accuracy Loss in Approximate Computing. In *in Workshop on Duplicating, Deconstructing and Debunking (WDDD)*.

[5] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proc. of the 31st Conf. on Programming Language Design and Implementation*.

[6] N. Barrow-Williams, C. Fensch, and S. Moore. 2009. A communication characterisation of Splash-2 and Parsec. In *IEEE International Symposium on Workload Characterization (IISWC)*.

[7] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, Peter Kogge, R. Stanley Williams, and Katherine Yelick. 2008. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead.

[8] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* (2011).

[9] William J. Bolosky and Michael L. Scott. 1993. False Sharing and Its Effect on Shared Memory Performance. In *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4.*

[10] Brett Boston, Zoe Gong, and Michael Carbin. 2018. Leto: Verifying Application-Specific Hardware Fault Tolerance with Programmable Execution Models. *Proc. ACM Program. Lang.* OOPSLA (2018).

[11] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proc. of the 9th USENIX Conference on Operating Systems Design and Implementation.*

[12] Milind Chabbi, Shasha Wen, and Xu Liu. 2018. Featherlight On-the-fly False-sharing Detection. In *Proc. of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.*

[13] L. Cheng, J. B. Carter, and D. Dai. 2007. An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing. In *IEEE 13th Int'l Symposium on High Performance Computer Architecture.*

[14] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. 2013. Analysis and characterization of inherent application resilience for approximate computing. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC).*

[15] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C. Chou. 2011. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *2011 International Conference on Parallel Architectures and Compilation Techniques.*

[16] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. 2012. Architecture Support for Disciplined Approximate Programming. In *Proc. of the 17th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems.*

[17] B. Grigorian and G. Reinman. 2014. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *2014 NASA/ESA Conference on Adaptive Hardware and Systems (AHS).*

[18] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. 1992. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessor. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[19] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. 2011. Dynamic Knobs for Responsive Power-aware Computing. In *Proc. of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems.*

[20] M. Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *IEEE Int'l Solid-State Circuits Conference Digest of Technical Papers (ISSCC).*

[21] Jaehyuk Huh, Jichuan Chang, Doug Burger, and Gurindar S. Sohi. 2004. Coherence Decoupling: Making Use of Incoherence. In *Proc. of the 11th Int'l Conference on Architectural Support for Programming Languages and Operating Systems.*

[22] S. Kaxiras and C. Young. 2000. Coherence communication prediction in shared-memory multiprocessors. In *Proc. 6th International Symposium on High-Performance Computer Architecture.*

[23] A. Kayi, O. Serres, and T. El-Ghazawi. 2015. Adaptive Cache Coherence Mechanisms with Producer-Consumer Sharing Optimization for Chip Multiprocessors. *IEEE Trans. Comput.* (2015).

[24] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. 2015. Rumba: An online quality management system for approximate computing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA).*

[25] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke. 2016. Quality Control for Approximate Accelerators by Error Prediction. *IEEE Design Test* 33 (2016).

[26] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. 1994. The Stanford FLASH multiprocessor. In *Proc. of 21 International Symposium on Computer Architecture.*

[27] Kevin M. Lepak and Mikko H. Lipasti. 2000. On the Value Locality of Store Instructions. In *Proc. of the 27th Int'l Symposium on Computer Architecture.*

[28] K. M. Lepak and M. H. Lipasti. 2000. Silent stores for free. In *Proc. 33rd Annual IEEE/ACM Int'l Symposium on Microarchitecture.*

[29] Kevin M. Lepak and Mikko H. Lipasti. 2002. Temporally Silent Stores. In *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems.*

[30] M. H. Lipasti and J. P. Shen. 1996. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Int'l Symposium on Microarchitecture.*

[31] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value Locality and Load Value Prediction. In *Proc. of the 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems.*

[32] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti. 2016. LASER: Light, Accurate Sharing dEtection and Repair. In *IEEE Int'l Symposium on High Performance Computer Architecture.*

[33] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman Jouppi. 2009. Cacti 6.0: A tool to model large caches. *HP Laboratories* (2009).

[34] Mihir Nanavati, Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. 2013. Whose Cache Line is It Anyway?: Operating System Support for Live Detection and Repair of False Sharing. In *Proc. of the 8th ACM European Conference on Computer Systems.*

[35] J. D. Owens, W. J. Dally, R. Ho, D. N. Jayasimha, S. W. Keckler, and L. Peh. 2007. Research Challenges for On-Chip Interconnection Networks. *IEEE Micro* (2007).

[36] David A. Patterson and John L. Hennessy. 2013. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface* (5th ed.). Morgan Kaufmann Publishers Inc.

[37] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization *(POPL '11).*

[38] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proc. IEEE 13th Int'l Symp. on High Performance Computer Architecture.*

[39] A. Raynaud, Zheng Zhang, and J. Torrellas. 1996. Distance-adaptive update protocols for scalable shared-memory multiprocessors. In *Proc. 2nd Int'l Symposium on High-Performance Computer Architecture.*

[40] S. K. Reinhardt, J. R. Larus, and D. A. Wood. 1994. Tempest and Typhoon: user-level shared memory. In *Proc. of 21th Int'l Symp. on Computer Architecture.*

[41] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with Relaxed Synchronization. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12).*

[42] P. V. Rengasamy, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das. 2015. Exploiting Staleness for Approximating Loads on CMPs. In *International Conference on Parallel Architecture and Compilation (PACT).*

[43] Michael Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. 2015. Monitoring and Debugging the Quality of Results in Approximate Programs. In *Proc. of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems.*

[44] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. 2014. ASAC: Automatic Sensitivity Analysis for Approximate Computing. In *Proc. of the 2014 Conf. on Languages, Compilers and Tools for Embedded Systems.*

[45] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based Approximation for Data Parallel Applications. In *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems.*

[46] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[47] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. In *Proc. of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[48] J. San Miguel, J. Albericio, N. Enright Jerger, and A. Jaleel. 2016. The Bunker Cache for spatio-value approximation. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[49] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger. 2015. Doppelgänger: A cache for approximate computing. In *48th Annual IEEE/ACM Int'l Symposium on Microarchitecture (MICRO).*

[50] J. San Miguel, M. Badr, and N. Enright Jerger. 2014. Load Value Approximation. In *47th Annual IEEE/ACM International Symposium on Microarchitecture.*

[51] Xiaowei Shen, Arvind, and Larry Rudolph. 1999. CACHET: An Adaptive Cache Coherence Protocol for Distributed Shared-Memory Systems. In *Proceedings of the 13th International Conference on Supercomputing (ICS '99).*

[52] Per Stenström, Mats Brorsson, and Lars Sandberg. 1993. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proc. of the 20th Annual International Symposium on Computer Architecture.*

[53] C. Sun, C. O. Chen, G. Kurian, L. Wei, J. Miller, A. Agarwal, L. Peh, and V. Stojanovic. 2012. DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling. In *IEEE/ACM 6th International Symposium on Networks-on-Chip.*

[54] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO 2020).*

[55] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. 2013. Quality programmable vector processors for approximate computing. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[56] S. Venkataramani, K. Roy, and A. Raghunathan. 2013. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *Design, Automation Test in Europe Conference Exhibition (DATE).*

[57] D. Wong, N. S. Kim, and M. Annavaram. 2016. Approximating warps with intra-warp operand value similarity. In *IEEE International Symposium on High Performance Computer Architecture.*

[58] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran. 2017. AxBench: A Multiplatform Benchmark Suite for Approximate Computing. *IEEE Design Test* (2017).