Exploiting Staleness for Approximating Loads on CMPs

Prasanna Venkatesh Rengasamy, Anand Sivasubramaniam, Mahmut Kandemir and Chita R. Das
The Pennsylvania State University
Email: {pur128,anand,kandemir,das}@cse.psu.edu

***Abstract*—Coherence misses are an important factor in limiting the scalability of multi-threaded shared memory applications on chip multiprocessors (CMPs) that are envisaged to contain dozens of cores in the imminent future. This paper proposes a novel approach to tackling this problem by leveraging the growingly important paradigm of approximate computing. Many applications are either tolerant to slight errors in the output or if stringent, have in-built resiliency to tolerate some errors in the execution. The approximate computing paradigm suggests breaking conventional barriers of mandating stringent correctness on the hardware, allowing more flexibility in the performance-power-reliability design space. Taking the multi-threaded applications in the SPLASH-2 benchmark suite, we note that nearly all these applications have such inherent resiliency and/or tolerance to slight errors in the output. Based on this observation, we propose to approximate coherence-related load misses by returning stale values, i.e., the version at the time of the invalidation. We show that returning such values from the invalidated lines already present in d-L1 offers only limited scope for improvement since those lines get evicted fairly soon due to the high pressure on d-L1. Instead, we propose a very small (8 lines) Stale Victim Cache (SVC), to hold such lines upon d-L1 eviction. While this does offer significant improvement, there is the possibility of data getting very stale in such a structure, making it highly sensitive to the choice of what data to keep, and for how long. To address these concerns, we propose to time-out these lines from the SVC to limit their staleness in a mechanism called SVC+TB. We show that SVC+TB provides as much as 28.6% speedup in some SPLASH-2 applications, with an average speedup between 10-15% across the entire suite, becoming comparable to an ideal execution that does not incur coherence misses. Further, the consequent approximations have little impact on the correctness, allowing all of them to complete. There were no errors, because of inherent application resilience, in eleven applications, and the maximum error was at most 0.08% across the entire suite.**

***Keywords*-Approximate Computing, Coherence, Caches.**

I. Introduction

Parallelism has become the de-facto standard for leveraging the continued transistor growth offered by Moore's law, while being confined within the power wall imposed by the end of Dennard scaling. The past decade has witnessed a proliferation of multiple cores on processors, and the numbers are expected to reach several dozen cores on a single die in the next few years. However, it is not clear whether the software can readily take advantage of this growing trend, and scale with the number of cores. A rigid contract at the hardware-software boundary is one of the impediments to such scalability, with hardware needing to strictly adhere to such contracts even if the application software is resilient to marginal violations. The area of *approximate computing* is drawing increasing interest to make the hardware more flexible, allowing some leeway in how the hardware operates assuming that the end results at the software level are within acceptable margins. This paper examines one such hardware contract - the need to fetch the latest version of shared data - from the approximation viewpoint, and shows that a stale version of data can be used in such cases to provide substantial performance gains without compromising on application accuracies.

As the number of cores increases, one of the scalability concerns is the consequent coherence actions across cached copies to ensure that a read from one core returns the value of the latest write (possibly written at another core). This consistency contract between the hardware and software, referred to as Sequential Consistency, has been the topic of much research over several decades to address coherence maintenance overheads. In addition to coherence protocol optimizations (e.g., [1], [2], [3]), relaxed consistency models have also been proposed (e.g., [4], [5]).Many of these relaxed models leverage synchronization points in the program to ensure that updated data values are propagated everywhere, allowing windows of overlap between the computation and coherence actions. However, it is not clear if all applications can be written that way, or whether all applications can really benefit (as some other studies have noted [6], [7]). To date, nearly all available application codes are based on sequential consistency, even if the underlying hardware allows some slight flexibility in the memory model.

An alternate emerging paradigm - approximate computing [8], [9], [10]- has not been well-explored in this context, and that is the void this paper intends to fill. The basic premise behind this paradigm is that either (i) applications have in-built resiliency so that any minor deviations from the contract by the hardware could be tolerated to still give the right/accurate output, and/or (ii) the accuracy semantics of the output is itself lenient to allow a range, rather than a fixed value (e.g., as in an image rendering algorithm where slight deviations in colors of a few pixels may be acceptable). Approximate computing has gained a lot of recent interest, with studies demonstrating benefits to specific applications (e.g., [11], [12]), and work on programming (e.g., [13], [10]), compilers (e.g., [14], [15]), runtime (e.g., [16], [17]) and hardware support (e.g., [18], [19]) for leveraging this paradigm. In this paper, we explore leveraging approximations to address the coherence overheads across the caches of a chip multiprocessor (CMP).

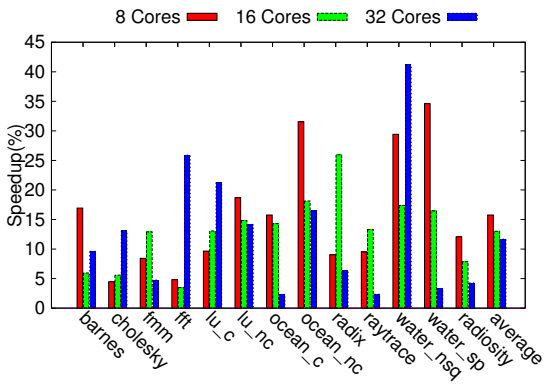


Fig. 1: Speedup by forwarding latest data values with zero cost to coherence load misses.

One of the important coherence actions in an invalidation-based protocol (MOESI and its variants that are used in cache-coherent CMPs) is the invalidate upon a write from a core, which wipes out all other copies. Even if the write cost can be hidden in the background, the subsequent costs of a load miss to those invalidated lines are usually in the critical path. Figure 1 shows the potential speedup that we would get for the SPLASH-2 benchmarks if we could avoid such load misses to invalidated lines, and if the cores could get the latest version of those lines without any coherence miss latencies (capacity and conflict misses are still incurred). We can observe improvements of up to 40% in some applications, with a 10-15% speedup on the average. This observation serves as a primary motivation for approximation in our study.

While one could try to predict values (as approximations) for such coherence related load misses as in some recent studies [20] (which would require additionally large prediction structures that are both power and area consuming), we propose to explore a simpler alternative - *can we return the (stale) value at the time of the invalidate to the subsequent load upon a miss?* Essentially, we are trying to leverage a stale value for such loads, with the presumption that either there is enough resiliency in-built into these applications and/or the results would still be within tolerable ranges/limits. Our secondary motivation for this work stems from the in-built resiliency features in many of the SPLASH-2 applications that are studied in this paper. As will be discussed later, many of these applications have several structures that are ideally suited for this type of approximation: (i) Some of them are convergence based where threads iteratively refine a solution. Since values do not radically change from one iteration to the next, using a value from prior iterations can still help the computations to proceed and even converge to the exact solution; (ii) Features such as smoothening/histogramming further allow computations to proceed based on ranges of values rather than a single value; and (iii) Some marginal errors in the outputs could still be acceptable (e.g., positions of bodies in a n-body simulation).

With these motivations, this paper proposes and evaluates mechanisms to approximate coherence-related load misses with *stale* values and studies their effect using detailed complete system simulations across the entire SPLASH-2 benchmark suite [21]. Along the way, it makes the following specific contributions:

- The first and obvious choice for storing these stale lines is the d-L1 cache itself where the line was invalidated, i.e., retaining those lines till they get evicted. We show that while such a scheme does not require additional hardware, these invalidated lines are likely to get evicted fairly soon, restricting their scope for serving those load misses.
- Instead, we propose a small (8 line) cache to hold such evicted invalid lines, which we call the *Stale Victim Cache (SVC)*. The coherence load misses could consequently get serviced by a stale line from either the d-L1 or the SVC, if present. However, with the SVC, there is a danger of letting such lines linger for too long, and getting more stale over time. When applications have several writes from other cores between two successive reads by a core, we need to be very careful about such staleness impacting the application (either performance or accuracy). Both issues - right sizing the SVC, and figuring out what stale data to retain in it - require careful consideration. We show that while the SVC provides higher speedup than by restricting ourselves to the d-L1 space, it still only gives half the potential suggested in Figure 1.
- To address SVC limitations, we set a time threshold to limit the staleness of data lingering in the SVC. The resulting SVC+TB (Stale Victim Cache with Time Bound) has the advantage of having more space in the SVC than the d-L1 alone to hold stale lines. Further, it limits the staleness of such lines, ensuring that this small space is effectively used for the less stale lines. We show that SVC+TB is able to provide the 10-15% speedup potential that is offered by the Ideal scheme of Figure 1.
- In the enhancements/approximations provided across all these mechanisms, we note that all SPLASH-2 benchmarks ran to completion, without crashing. Further, the inherent resiliency in these applications ensured that 11 of the 13 benchmarks gave outputs that 100% matched the results of the non-approximated execution. Even in the 2 benchmarks where the outputs differed, the errors were at most 0.08% in the worst case.

Thus, this paper adapts victim cache [22] to approximate computing. It also demonstrates that in-built resilience to approximations found in parallel applications can reap performance benefits without much/any loss in accuracy.

To our knowledge, this is the first paper that studies the effect of staleness-based approximation for coherence-induced load misses for the widely used SPLASH-2 benchmark suite. Our intent, in this paper, is not to suggest that such a staleness based approximation to coherence misses be used across all applications, or across the entire memory space. Rather, we aim to show the potential of this approach in these widely studied shared memory multi-threaded applications, and explore architectural mechanisms to achieve this potential. Further, these approximations could be done for selective loads in an

application, with possible programmer annotations, compiler and/or runtime support for such selection. In this paper, we leave such issues for future work, and only consider doing these approximations at the entire application granularity (across all memory locations with coherence misses), except for the memory locations within the OS.

II. Related Work

Coherence Optimizations:: The area of cache coherence has witnessed numerous proposals and optimizations over the past decades. Invalidation-based protocols (variations of MOESI) are widely in use on most current CMP platforms [23]. Both directory costs (e.g., [24], [25], [26]) and coherence miss costs have come under a lot of scrutiny over the years. Recognizing the need for selective invalidation and update mechanisms, several hybrid alternatives have also been proposed (e.g., [1], [27]). Rather than devise a new protocol, our goal is to try to optimize for coherence misses within the confines of the existing MOESI.

Prefetching:: One such latency-tolerance technique which can work within the confines of a protocol is prefetching, which fetches lines ahead of their use. Prefetching can be implemented either in hardware, or in software, and both kinds (e.g., [28], [29]) have been exploited to hide miss latencies for multiprocessors. Prefetching, typically, needs additional structures for predicting addresses, unless they are for simple patterns like next line prefetching. Further, prefetching can sometimes worsen performance (e.g., [30]), especially with the limited on-chip and off-chip bandwidth that is shared across multiple cores of a CMP. Our solution does not preclude prefetching, and can complement any miss latency reduction that prefetching can offer.

Relaxed Consistency Models:: Another approach, as discussed in Section I, is supporting a weaker memory consistency model (e.g., [4], [5], [31]) in the underlying hardware and leverage application level constructs (synchronization, fences, etc.) to dispatch the requisite coherence actions. Despite numerous models being proposed, studies have shown that the same benefits could be attained through other means (e.g., [7]). Further, despite the opportunity of leveraging this capability, most application codes are still based on the sequential consistency model.

Speculative Coherence:: A closely related work [32] is based on the hypothesis that the invalidated lines in MOESI have not really changed (despite being invalid), and allows the load misses to read those lines and proceed speculatively. Concurrently, the validity of this hypothesis is tested, and if false, the computation is rolled back. Our work is also based on reading such invalidated lines. However, we do not roll back state, and the associated complexities of state maintenance, which can be substantial, are avoided. Further, we show that such lines should be held on to much longer than what the restricted space in the normal d-L1 would allow, as is done with our Stale Victim Cache. On the other hand, their work shows that such speculation is correct nearly 70% of the time, serving as a motivation for our work.

Victim Caches:: Victim caches have been mainly studied [22] for conflict misses, and have not been previously proposed to hold evicted invalidated lines for multiprocessor coherence. Further, our Stale Victim Cache also imposes a time bound for how long they reside within this structure, to limit their staleness.

Approximate Computing:: There has been work on the applications [11], [12], [33], programming [13], [10], compiler [14], [34], [15], runtime [16], [17] and hardware [18], [35], [19] fronts to support this paradigm. From this domain, the recent work on load value approximation [20] is the most relevant. By predicting load values, their work attempts to reduce miss latencies and allows computation to proceed without getting into the critical path. Their work is across all load misses, and not specific to coherence. However, in addition to depending on prediction accuracy, large structures (e.g., a 10KB to 18KB size structure is proposed in [20]) with their associated power and area overheads, are needed to make such predictions. In contrast, our approach requires a very small (8 line) structure, and is able to achieve nearly all of the potential speedups attained by short-circuiting coherence misses (shown in Figure 1).

Bounding Staleness:: Also, there has been some recent work from different domains such as storage systems, programming languages and big data focusing on bounding staleness of data [36], [37], [38], [39]. Our SVC+TB differs from them as we use bounded staleness for an entirely different purpose and consider a hardware-based implementation.

To our knowledge, this is the first work to attempt leveraging stale (invalidated) lines in the cache to approximate coherence load misses. In the process, it shows that augmenting the existing d-L1 with a very small Stale Victim Cache, can provide considerable benefits to mitigate these misses. It also shows that such an approximation has little impact on application accuracy for the SPLASH-2 benchmarks.

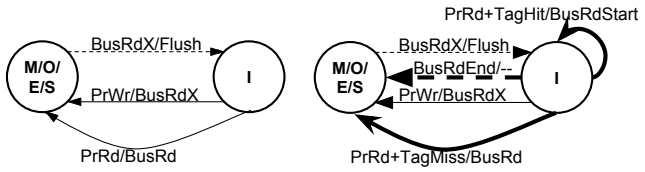
III. Mechanisms for Exploiting Staleness

Towards our goal of leveraging stale values that a core may have previously cached for serving subsequent loads, we explore two main strategies - (i) using space in the d-L1 cache itself, and (ii) supplementing the L1 with a small Stale Victim Cache (SVC) - that are described below.

A. Reading Invalidated Lines (RIL)

In the normal MOESI protocol, a cache line could be in Invalid state for one of the following two reasons: either (i) the line is not present in the cache at all (i.e., the tags will not match on a look up), or (ii) it is present in the cache but it has been invalidated (by another core's write) and the data is thus stale. Normal MOESI does not distinguish between these two possibilities, and treats all loads to such a line as misses, fetches the line from elsewhere in the hierarchy (another L1, or L2 or even main memory) and subsequently changes the state of the line appropriately as is denoted in Figure 2a.

In our first approach called RIL, we differentiate between these two possibilities. In RIL, MOESI functions as is to



(a) Original MOESI State Flow from MOES to I

(b) Approximate/Stale Read From an Invalid Line

Fig. 2: Changes to MOESI for reading an invalidated line indicated in **bold** lines: (a) shows all reads and writes to I lines to transition immediately to the appropriate MOES states. In (b), when PrRd occurs, if there is a TagHit, we supply the data and start a BusRd in parallel. When the BusRd Ends, we transition to the corresponding MOES state. However, if there is a TagMiss in I state, it follows the original transitions in (a).

service d-L1 Load misses if it is in case (i). The difference is in case (ii) where we return the (possibly stale) value that is present in the invalidated line in L1. Concurrently, this load miss is serviced in the background, fetching the latest version of the line from elsewhere in the memory hierarchy - either from another L1, or L2 or even as deep as main memory. The core is not stalled in the latter case, and proceeds as if the load is complete. When the latest version of the line is eventually brought in, the local state of that line in d-L1 would then be set in an appropriate state as determined by MOESI. While there has been some prior work to read the data (stale value) early from the invalidated state in L1 [32], their work still enforces correctness of each load by rolling back the state if the loaded value is different from the subsequently brought in latest version. RIL, on the other hand, does not roll back the state even if the values are different (in fact, it does not even check whether the values are different), and is hence an approximate execution. From the hardware viewpoint, implementing RIL only requires a minor MOESI modification as depicted in Figure 2b.

B. Stale Victim Cache (SVC)

While RIL is easy to implement and can reduce the load stall times of coherence misses when applications can tolerate data staleness, its potential is limited by the number of times that a load can indeed find the corresponding block in the cache in an Invalidated state. It is possible for such a line to get evicted (by the replacement policy) from the corresponding L1 set between the invalidation (by another core) and the load from the local core. In such cases, we do not even have the stale data to serve the load and will have to resort to the default MOESI actions.

As we will later show, the percentage of coherence misses that RIL can serve is only 2% on the average. This leaves a large gap of coherence misses, which cannot be approximated with stale values. This motivates the need for holding on to such invalidated (stale) lines longer than the residency time offered by the normal cache eviction policy. While one could develop and optimize a policy tailored for replacing such lines, we opt for a simpler solution - a victim cache - that has been tried [22] to address similar problems in the context of capacity/conflict misses of caches. The difference is that our victims are only those invalidated lines (by coherence actions) that are being evicted out of the L1, and we consequently use a *Stale Victim Cache (SVC)* to hold on to such lines. We propose to use a small SVC (8 line, fully associative that is justified in Section V-C2) in hardware for which we need to understand the following three actions:

- *Insertion:* When an (coherence) invalidated cache line is evicted from d-L1 by the normal cache replacement policy, it is inserted into the SVC. Any other d-L1 lines being evicted go through the normal eviction/write-back options.
- *Lookup:* Upon a load, SVC is looked up concurrently with the normal d-L1. If the load address tag matches in the normal d-L1, then the data is returned from there (whether valid or stale). Else, if the line is present in the SVC, then the corresponding data (it is definitely stale in this case) is returned and the load completes. Concurrently, as with RIL, MOESI actions are performed in the background to fetch the actual data from elsewhere in the hierarchy.
- *Eviction:* There are two ways by which an entry can get evicted from the SVC: (a) When the data is eventually brought in from deeper in the hierarchy with MOESI actions, after a stale version is returned back to the load, the line is no longer stale. It is then removed from SVC and moved back to d-L1 (in a valid state); (b) Because of capacity/conflict issues in the SVC, a line can get evicted to create space for another line's insertion. In such cases, the line to be evicted from the SVC is simply thrown out. Any subsequent load to that line will not find it either in d-L1 or the SVC, and will be served as a load miss by MOESI.

C. Stale Victim Cache with Time Bound (SVC+TB)

On the one hand, RIL may not be able to serve stale values for too long, since those invalidated lines are likely to get evicted from the high load imposed on d-L1. While the shorter window of staleness lessens the scope for approximation and its associated benefits, the advantage is that the detrimental effect of staleness is also limited. At the other end, the SVC mechanism described above can keep these stale values for a long time to extend the scope for approximation. However, as we shall later see in the results, having very stale values (i.e., serving successive loads to an address with the same old value, or returning the value to a load that was written by a much older store operation) can impact the execution in multiple ways. It can worsen application level errors, making results/correctness unacceptable. It can even worsen the execution due to several factors, e.g., applications requiring more computation/iterations to converge to a solution, waiting longer at synchronization points, etc.

To bound the staleness of data in SVC, we additionally explore a mechanism to *time-out* the residence of stale lines in the SVC with a simple thresholding mechanism. When a line is inserted into the SVC (i.e., invalidated line is evicted

from L1), we record the time-stamp in cycles. Whenever the lookup operation finds a line in the SVC, it compares the current time-stamp with the time-stamp of insertion. If the difference (elapsed time) exceeds a threshold, we simply treat it as a miss in the SVC, and can also throw it away from the SVC. We will discuss the impact of this threshold later in the paper and show that a time bound of 100 cycles is a good trade-off point for keeping some amount of stale data to serve loads, while not becoming overly stale.

We could potentially use a time bound for RIL as we do in SVC+TB. However, d-L1 usually has high hit rates and we found that this needed quick replacements to invalidated lines. Hence, we do not discuss the results for RIL+TB in this paper.

D. Software Issues

Reading stale values, to avoid load misses, may not always be a panacea. Apart from program correctness (and their crashing), staleness can have detrimental performance consequences. For instance, applications with iterative convergence may take more iterations to converge. Control flow changes can also result in more (or less) instructions to be executed. Further there are specific constructs where staleness may not help. For instance, consider spin-lock implementations where a core spins on a cached lock variable that has to be written by the current lock owner. With more reads (than writes in between), the staleness for such data is likely to be less. Even reading a staler version (which says that it is still locked) will only make the core continue spinning until the actual data value (saying unlocked) is returned - there are no correctness violations. Note that the hardware already optimizes for such scenarios with the built-in MSHR check to stop duplicate read requests from being issued. While there could be a lot of future work on selectively performing our optimizations (specific data structures in a program), in this work, we consider a rather simple approach - studying these mechanisms at the entire application granularity across the entire memory space. When an application is run from the command line, one could simply state whether to avail of our optimizations or not. We, however, do not approximate loads of privileged mode contexts (operating system code). We also do not allow stale value access for the address translation part of accesses which may have disastrous consequences if not accurate. It is our intent to explore whether SPLASH-2 benchmarks can benefit from our staleness-based approximations and if (and how much) accuracy is compromised, rather than when and where to use this approach (which is part of future work).

IV. Experimental setup

A. Hardware Platform

We use GEM5 [40] simulator in full-system mode for all our simulations. The parameters used are listed in Table I. Our simulations use a multicore CMP with Alpha ISA cores, running a MOESI protocol [23] across L1 caches to keep their blocks consistent. The L2 cache is shared across all the cores, and its banks are distributed in a S-NUCA organization [41]. Accesses to a L2 bank may, thus, require traversing the on-chip interconnect for which we use a 2-D mesh organization. L2 misses will then subsequently be re-directed to the appropriate memory controller for fetching those requisite cache blocks from main memory. On top of this architecture, our proposal considers a small SVC at each d-L1. We modify the MOESI protocol to allow reading of stale values, present either within the normal d-L1 or in the SVC, as discussed previously in Section III, upon a d-L1 miss while still initiating the actions in the background to serve this miss. We consider different configurations and replacement strategies for this small SVC.

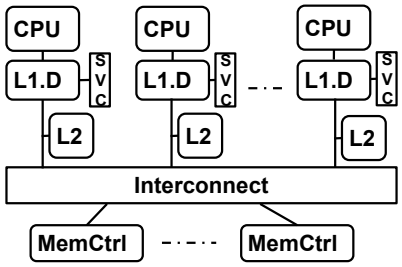


Fig. 3: Architecture used for simulations. SVC is our additional 8 line cache holding invalidated lines evicted from d-L1.

Parameter	Value
Cores	8, 16 and 32 Cores, out of order, ALPHA, 2.24GHz
LQ, SQ, ROB	32, 32, 192 (per core)
L1	32 KB I Cache, 32 KB D Cache (private per core) 2 Way Associative, 2 Cycles hit latency, 4 entry MSHR LRU replacement policy, MOESI coherence protocol
L2	128 KB per core, S-NUCA 8 Way Associative, 20 Cycle hit latency, 20 entry MSHR LRU, distributed directory
On-chip Network	Mesh $4 \times \{2, 4, 8\}$ 32B flits, XY routing, 16 buffers per router, 2.24GHz
Memory Controllers	2 (for 8 cores), 4 (for 16, 32 cores)
DRAM	2 Ranks per channel, 8 Banks per rank, 1KB Row buffer

TABLE I: Simulation parameters.

B. Workloads and Checkers

Since our optimizations are intended for multi-threaded applications that share memory, we have used multi-threaded applications from the SPLASH-2 suite [21]. We would like to note some key differences in our study from most prior architectural work (which do not approximate the execution) using such benchmarks, leading to the following implications: (i) Since we are approximating loads, it is not sufficient to only check the performance. We also need to *verify* the impact/accuracy of the approximations on the program's correctness; (ii) Not all programs have in-built, or readily available, mechanisms to check the correctness of the program's execution. This requires us to implement additional checkers to verify the program's output and report any errors when deviating from the correct execution; and (iii) Sampling/subsetting the overall execution does not suffice. While sampling/subsetting mechanisms are often used to speed-up the simulation/evaluation, it is not clear that such mecha-

Bench -mark	Accuracy Impacting		Performance Impacting			
	Method	Checkers Built-in?	Avg. Staleness	Coh. Miss%	d-L1 Miss%	Sync. %
Ocean-NC	Tolerance check	Yes	2.47	14.6	4.51	0.004
Raytrace	Smoothening	No	2.53	**6.1**	5.67	0.75
Radiosity	Tolerance check	Yes	2.54	**7.7**	6.58	0.8
Cholesky	Block mult + add	Yes	2.55	**9.6**	3.35	**4.2**
Barnes	Smoothening	No	2.57	**7.9**	4.45	0.0005
FMM	Smoothening	No	2.58	14.4	4.95	0.0005
Water Nsq	Tolerance check	No	2.58	12.7	1.43	0.002
Radix	Sort per hex digit	Yes	2.6	14.9	6.52	**8.97**
Ocean-C	Tolerance check	Yes	2.6	14.2	5.29	0.004
Water Sp	Tolerance check	No	2.61	12.5	1.57	0.61
LU-NC	Mult + add	Yes	2.7	13.1	6.01	0.0007
LU-C	Mult + add	Yes	2.81	13.4	4.16	0.0005
FFT	Mult + add	Yes	2.96	14.7	5.59	0.0006

TABLE II: Characteristics of SPLASH-2 benchmarks impacting staleness tolerance and associated performance benefits. Numbers are based on Baseline execution on a 32 core configuration without any approximated/stale loads. Applications are sorted based on Staleness which is Avg. of number of writes between successive reads to invalidated lines (and has to be ≥ 1). Coh. Miss% = Coherence Misses / Load Misses in %. d-L1 Miss% = d-L1 Misses / d-L1 Accesses in %. Sync% = Accesses to Sync Data / Accesses to shared Data in %. Bold values show outliers with low Coh. Miss% and higher Sync. %.

nisms capture the effect (and propagation) of inaccurate loads. Consequently, we need to simulate each application from beginning to end, without skipping any instructions, leading to really long simulation times.

The left half (Accuracy Impacting) of Table II gives some characteristics of these benchmarks, in terms of their main operations which may be resilient to approximations/staleness, and the availability of checkers for verifying correctness of results. Many of them have some in-built resiliency to handle errors arising from staleness as discussed below.

- Except Radix, all other applications operate on floating point data for most of their important computations. Note that floating point numbers are already approximated/rounded in machine represented binary form. More importantly, many of these algorithms use smoothening functions on their input float values, allowing ranges of values for these numbers instead of a single fixed value. For instance, in FMM, it computes interactions between all pairs of particles in a thread in each iteration. During this computation, if two particles are less than a $threshold$ units apart, then their relative positions are assumed to be exactly $threshold$ units. It then uses this position to update the particle's force's magnitude and direction. This option gives us the opportunity for the particle's position to deviate between $-threshold$ to $+threshold$ without any accuracy loss.
- Further, many of these applications (e.g., Ocean, Radiosity and Water) use iterative convergence computations, where the solutions are iteratively improved upon (in time steps), till they stop changing. In such cases, stale data values could still let the computation proceed, even if the number of iterations themselves could take longer to converge. Many of the other applications also use such iterative improvements on solutions, with the difference being the exit condition (e.g., a fixed number of time steps is used in Barnes and FMM).
- Large data sizes can also make these applications resilient to approximations. For example, in the Water benchmarks, the computed result is the potential energy of a body of 512 molecules as a single floating point number. Even if one molecule's position or momentum does not reflect the up-to-date value, it will have negligible impact on the overall result.

The above mentioned points are only to give the reader an indication of why staleness of data may still be acceptable in these applications. Regardless, we still need to verify the accuracy of the computation at the end of the execution, and we call such verification mechanisms as *checkers*. Checkers, are obviously, not universal and need to be custom written for each application. In 5 of the benchmarks (Cholesky, FFT, LU, Ocean and Radix), the applications already have in-built checkers and we simply report the errors calculated by those checkers with our approximation. In each of the other benchmarks, we have custom-written checkers as is summarized in Table III. Apart from a numerical value for an error, note that the result of an approximate run could be as bad as making the entire program crash/hang.

C. Metrics

In our subsequent results, we will use the following metrics to study the impact of staleness when approximating loads in these applications:

- **Speedup (%):** Rather than employing metrics such as IPC, which may not capture the fact that total instructions executed may not necessarily be the same across different runs, we show speedup (as a percentage) of the approximated run with respect to the accurate run. Given that we run each application from beginning to end, the speedup is a good measure of the performance improvement.
- **Error (%):** Correspondingly, we show any loss in accuracy (as a percentage of the accurate run) in results with the approximate run (as indicated in Table III). If this is a finite value, then the approximate execution did complete, albeit with erroneous results. Else, the program crashed/hung.
- **Increase in Instructions (%):** An approximate run could result in higher number of instructions executed. Possible reasons include higher number of iterations in convergence, changes in control flow and/or even due to changes in synchronization events (e.g., number of unsuccessful spinlock attempts). We show this percentage separately, whether or not these instructions resulted in a overall net slowdown or the execution was sped up despite these additional instructions.

V. Results

We compare the 3 schemes in Section III with the Ideal execution explained in Section I. Ideal assumes that there

Program	Correctness Metric
Barnes	Simulates N-body interaction in 3D space. It outputs position, velocity, acceleration and potential of each particle after simulating N steps. We compare the output parameters of our run with the accurate run, and report mean and standard deviation.
FMM	Computes position of n charged particles in an XY plane. We compute the relative distance between all $\binom{n}{2}$ pairs of particles and compare them with that of the accurate run. Accuracy is the average of % change in the relative distances from the accurate run.
Radiosity	Computes equilibrium distribution of light (RGB components) in a scene. We check accuracy as the euclidean deviation of the final equilibria of RGB values of both accurate and approximate runs.
Raytrace	Renders a 3D object on 8 2D images. We check accuracy as the average Euclidean deviation between the pixels of the output images in accurate and approximate runs
Water	Computes the potential energy of n water molecules. The output is the force exerted by each atom in XYZ directions. We check accuracy as the resultant force between all pairs of $\binom{n}{2}$ molecules, and compare with that of the accurate run to report deviation.
Cholesky, FFT, LU, Ocean and Radix have inbuilt check mechanisms. We simply report the error percentage computed by these checks.	

TABLE III: Accuracy checks for SPLASH-2 benchmarks.

are no load miss costs for coherence misses, and there is no staleness in the data read by the consuming core either, and is thus a highly optimistic target. Before getting into the results for each mechanism, we first highlight some of the important application characteristics that would impact their ability to leverage staleness.

Access String	$R_{0,A}$	$W_{1,A}$	$R_{0,A}$	$W_{3,A}$	$W_{2,A}$	$R_{1,A}$	$R_{0,B}$	$W_{3,B}$	$R_{0,B}$
Staleness	-	-	1	-	-	2	-	-	1

TABLE IV: Example showing the computation of Staleness. The requests are shown in increasing time order from the left to the right.

A. Application Characteristics Affecting Performance

The right half (Performance Impacting) of Table II summarizes some representative characteristics of these applications, that can help understand the impact of leveraging staleness in subsequent results. These characteristics have been obtained from their respective "default" executions on a 32 core system, which do not perform any approximations and all coherence misses will necessarily go and fetch the right version of the data to satisfy the miss. Specifically, we note four attributes in these applications to understand the impact of staleness:

- *Avg. Staleness*: A coherence related load miss at a core can read a stale value because there is at least one write/store since the last time that the data was brought in by the load from the core. It is possible that there are multiple such stores between these successive loads, implying that the latter load could see a much staler value if it does not receive the values written by the stores. The number of such stores between these successive loads is a reasonable representation of the staleness of the data leveraged by our approximations. The column titled "Avg. Staleness" in Table II shows this number, that is averaged across all loads that incur coherence misses. We give a small example to explain the computation of Avg. staleness in Table IV. We use $\mathrm{R}_{\langle Core\rangle,\langle Block\rangle}$ or $\mathrm{W}_{\langle Core\rangle,\langle Block\rangle}$ to represent a read or write access from the $Core$ to this $Block$ respectively. Recall that, Avg. Staleness is defined as the number of remote writes between two successive reads to a shared block. In this example, there are three such shared reads. Therefore, Avg. Staleness of this access string is $\frac{1+2+1}{3} = 1.3$. Note that the applications in Table II have been sorted based on this value, increasing in such inherent staleness as we go down the rows.
  In general, this attribute (and consequently the sorted order) captures the possible extent of the staleness that a scheme may relatively cause. With a higher number, the same scheme may serve a much staler value of a data item than when the number is lower. For instance, the same scheme may return a much staler version of the data to a subsequent load in FFT (with a value of 2.96) compared to a load in say Ocean_NC (with a value of 2.47). Such staleness, apart from possible accuracy issues, could also impact other control flow decisions, e.g., taking more iterations to converge with staler values. Staleness is expected to have a more significant impact on SVC (which does not put a specific bound on holding on to those values in the SVC), compared to RIL (where they can get more easily evicted from d-L1) or SVC+TB (where we have a time bound).
- *Coherence Misses (%)*: Despite what the above metric may indicate, an application is not going to benefit from the staleness approximation from the performance viewpoint if the percentage of coherence misses is only a small fraction of the total load misses. As can be seen, despite having average staleness less than 2.6 (which is significantly lower than FFT or LU), Raytrace (6.1%), Radiosity (7.7%), Cholesky (9.6%) and Barnes (7.9%) have low coherence misses (less than 10%), suggesting that they are not going to benefit significantly from such approximations.
- *d-L1 Misses (%)*: This measure captures the total pressure on the valuable d-L1 space. Higher miss rates (which include capacity and conflict misses, apart from coherence misses) can cause more rapid eviction of the invalidated/stale lines. That can reduce the effective d-L1 space available to hold stale values. It can also put more pressure on SVC if the evicted lines are coherence-invalidated lines.
- *Sync. Accesses (%)*: Another factor that can go against the trend suggested by the above mentioned staleness metric is the synchronization in the program. Consider a spinlock where a core is checking the status of the currently held lock by performing loads of the lock variable, which would get cached. In general, such accesses would incur fewer stores (the release of the lock) since the last invalidate before the line is read again, leading to a lower staleness

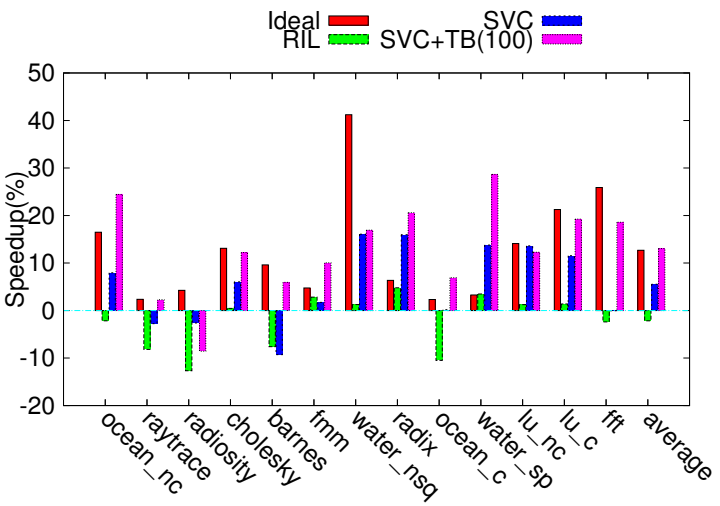
metric. However, in such cases, approximating the variable with a stale value does not really serve a purpose from the performance viewpoint since the core would simply keep looping till the lock is detected to be free. Hence, examining the number of synchronization accesses (as a percentage of shared accesses) can help us understand whether the staleness based approximation is indeed beneficial. For instance, even though Cholesky and Radix have relatively lower staleness metrics (less than 2.6), they have a considerable number (4.2% and 8.97% compared to less than 1% in others) of shared accesses going to synchronization variables, suggesting that a staleness based approximation may not be as useful for them.

B. *Effectiveness of Our Mechanisms*

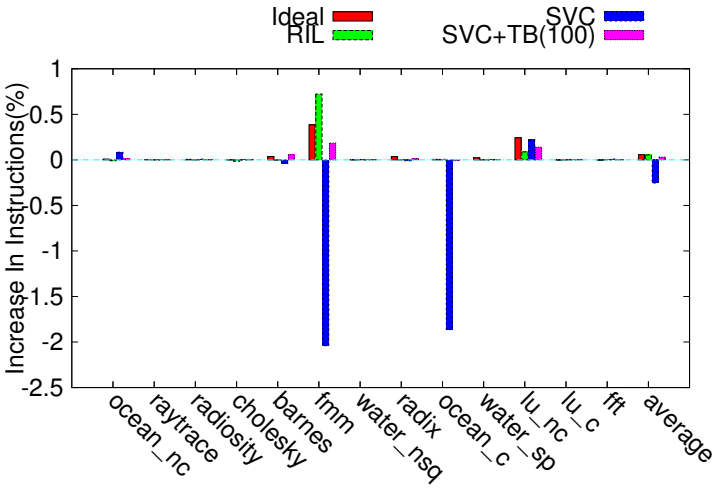
In the interest of space and clarity, we first specifically focus on 32 core configurations with (i) RIL, (ii) an SVC configuration that uses a very small 8-line, 4-way associative stale victim cache, and (iii) an SVC+TB(100) configuration that uses a 8-line, fully associative stale victim cache with a time bound of 100 cycles. We have also conducted experiments with other core and SVC configurations, and the results are summarized later in Section V-C2. The three schemes are compared with Ideal, where the data is automatically forwarded upon a coherence miss to the load without incurring any latency. The results are shown for each application, together with the average across all applications, in Figure 4 in terms of the speedup %, increase in the number of instructions, and Error % metrics. The applications have been ordered in the same way as in Table II (increasing Avg. Staleness order).

Staleness-based Approximation Benefits across Applications:: While having lower Avg. Staleness values (say < 2.6) can help applications as was mentioned earlier in Section V-A, there are other counter-acting forces. Four of the applications - Raytrace, Radiosity, Cholesky and Barnes - do not have as many (less than $< 10\%$) coherence misses compared to the others and hence may not have as many opportunities to benefit from converting them to misses, as is evidenced by their lower speedups (of 2.4%, 4.3%, 13.1% and 9.6% respectively) even in the Ideal case. Cholesky additionally has several synchronization instructions, and as described earlier, such instructions may not benefit from staleness, or from avoiding the misses even if the data is not stale (i.e., Ideal execution). In the case of Radix, even though the number of coherence misses is relatively high (14.9%), it has the highest percentage of synchronization instructions (8.97%), again impeding its speedup (6.4%). Consequently, the average speedup of these 5 applications (Raytrace, Radiosity, Cholesky, Barnes and Radix) is only 7.1% in the Ideal execution, which is relatively lower than the 16.2% speedup averaged across the rest.

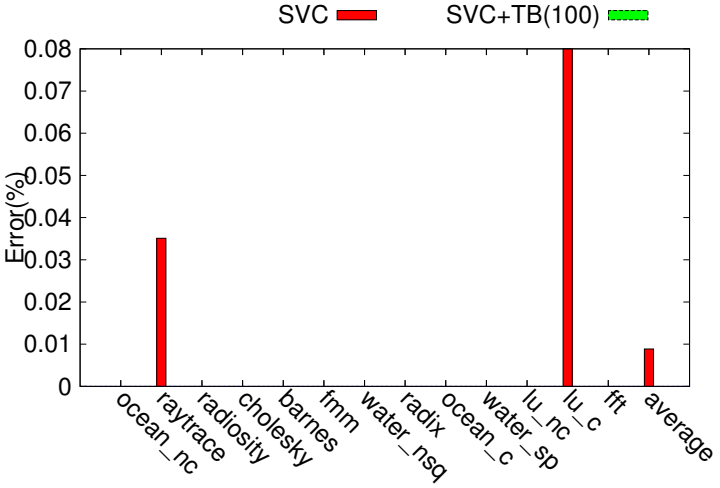
Ideal Does Not Gain Maximum Speedup:: It is to be noted that Ideal does not necessarily mean the best attainable speedup for every application with the staleness based approximation. Ideal saves only the latency of fetching the correct data, but it does NOT deviate from the original control flow. But control flow can be highly data dependent, and it is possible for some of the staleness-based executions to skip in-



(a) Speedup



(b) Increase in Instructions



(c) Accuracy

Fig. 4: Results with 32 cores. SVC is 8 lines, with 4-way assoc. SVC+TB(100) is 8 lines and fully associative.

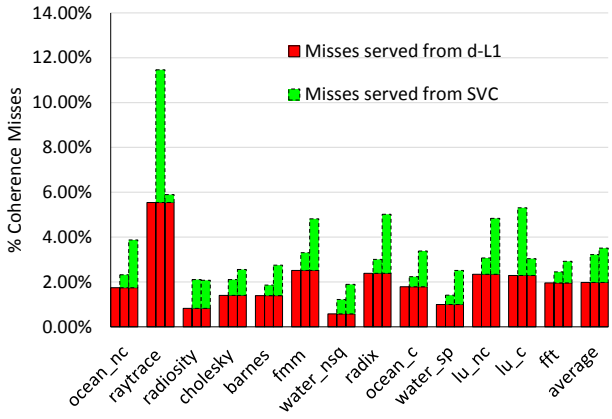


Fig. 5: % Coherence misses fulfilled by stale values in the 3 mechanisms. Each group of 3 bars correspond to RIL, SVC and SVC+TB(100) respectively. All 3 could serve stale data from d-L1, while the latter two could additionally serve from their SVC.

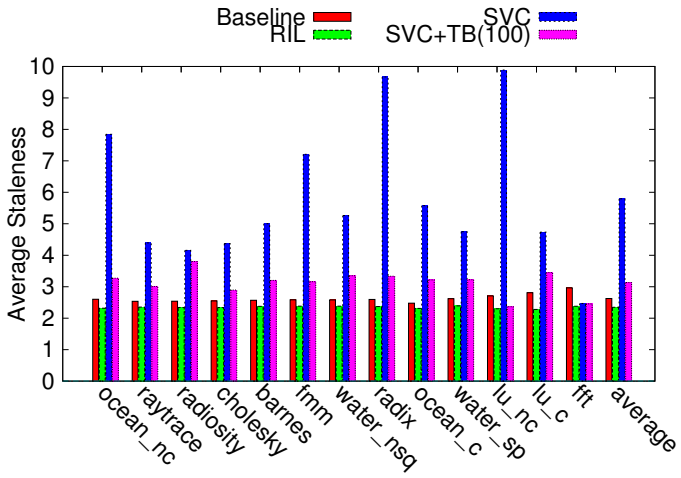


Fig. 6: Avg. staleness of data supplied. RIL has a lower probability of supplying stale data and hence has a lower delivered staleness. In SVC, there is a danger of serving very stale data. SVC+TB trades-off these two extremes.

structions similar to loop perforation [14], and not necessarily require executing more instructions to complete. Note however that our solution may be applicable even beyond loops like lock constructs (increase in instructions with a risk of spinning longer than the accurate control flow), value smoothening (hiding load latency without any accuracy penalties) etc. We see evidence of this in Figure 4b, where in some cases we incur a higher number of instructions, and in some cases a lower number, compared to the default execution. *Note the scale of this graph, which ranges between -2% to 1%*, indicating that any such variation in the number of instructions is rather negligible.

Comparing the Mechanisms:: We now compare the results of the three staleness-based approximation mechanisms in Figure 4. To further understand these results, we use (a) Figure 5 which shows the percentage of coherence misses that are serviced by a stale value in each of the three schemes, and (b) Figure 6 which shows the average staleness of data supplied by each of these schemes. Note that in RIL, the stale value can be supplied only if it is present in the normal d-L1. On the other hand, both SVC mechanisms could serve this either from d-L1 (if present), or from their respective SVCs. Consequently, in both SVC and SVC+TB(100), we give this breakdown when serving coherence misses with stale values.

RIL is not able to benefit these applications. It gives speedups in only six executions - FMM, Water-Nsq, Radix, Water-Sp, LU-NC, LU-C - and that too with only an average speedup of 2.5% in these six. Considerable slowdowns are incurred with RIL in Raytrace, Radiosity, Barnes, and Ocean_C. On the average, RIL results in a 2.15% slowdown across these benchmarks.

When examining the coherence misses serviced by RIL in Figure 5, we note that it is only able to service 61% of the total coherence misses serviced by SVC on the average. Since RIL is limited by the opportunistic space offered within the existing d-L1, it may not be possible to hold a lot of such stale lines. Normally high capacity and conflict misses in d-L1 would quickly evict such stale lines, making RIL resort to the default way of servicing a load miss. For instance, consider a high d-L1 miss rate application such as LU_C (4.16%). Despite having a reasonably high coherence miss rate, RIL is able to supply only half the stale values that the SVC approach can provide, due to their eviction from d-L1 which becomes more pronounced with higher d-L1 pressure. To add to this observation, Avg. Staleness in this application is relatively higher (2.81), suggesting that successive reads to those shared lines from a core are more spaced out. This gives those stale lines a higher chance of being evicted from d-L1 in RIL.

On the average, RIL is able to only service around 2% of the coherence misses with stale values. These results suggest that simply using stale data from the d-L1 does not suffice. Instead, we need to hold on to such data longer as in the SVC schemes. In the standard SVC results, we see an overall improvement, with an average speedup of 5.5% over the baseline across all the benchmarks, which is substantially better than the slowdown obtained with RIL. SVC is generally better than RIL in nearly all applications, with the notable exceptions being Barnes and FMM. While pre-maturely evicting the stale lines from d-L1 in RIL gives less opportunity when servicing coherence misses, it may have the advantage of limiting the amount of staleness in these lines. Compare this with a SVC, where such lines may linger in the alternate SVC structure, serving coherence misses for a lot longer compared to RIL. For instance, in Barnes, RIL was itself doing badly with a slowdown of 7.6%, serving stale data for only 1.4% of the coherence misses. When running on SVC, the performance of the same benchmark worsens even more (slowing down by 9.3%) despite serving 33% more coherence misses than RIL. The staleness of the data has significantly worsened, leading to this deterioration. On the average, the staleness of data served

by the SVC mechanism has more than doubled over that served in RIL (Figure 6).

SVC+TB(100) tries to attain the relative benefits of both RIL and SVC, without their individual drawbacks. On the one hand, it is not limited in capacity by the opportunistic space (before eviction) in d-L1 as in RIL. On the other hand, despite having a separate SVC, it bounds the residency within this structure to limit the staleness (Figure 6). This is very apparent when we examine the result for Raytrace. As was shown earlier in Table II, Raytrace does not have high enough coherence misses to benefit tremendously from avoiding them through stale values. With a high enough d-L1 miss rate, there is only a 5.67% chance of finding a stale version in RIL. SVC on the other hand, has more than doubled this chance by returning as many stale values from the stale victim cache (Figure 5). However, with this lower coherence miss rate, such stale lines could persist for a long time within the SVC, without getting evicted (since capacity/conflict misses in the SVC itself would be lower). We see that despite this doubling of serving stale values, SVC does not really give any performance benefit (it is in fact a slow down) in Figure 4a. On the other hand, with SVC+TB(100), even though it does not approximate as many loads as SVC (and only slightly higher approximations than RIL), it does not have the deficiencies of these two approaches and actually provides a 2.28% speedup over the baseline. Overall, SVC+TB(100) gives 13% speedup on the average across all benchmarks, becoming comparable to the Ideal executions.

Accuracy:: As explained in Section IV, we have also tracked the accuracy of the results of the application run with the approximated loads in each of the schemes, and the error % is shown in Figure 4c.We first note that none of the executions crashed, and they all ran to completion. Further, in all but two (Raytrace and LU_C), the results matched those of the accurate run, giving no errors. Even in those two with errors, the error percentages are extremely low (less than 0.1%). These results clearly demonstrate the inherent resiliency of these benchmarks to tolerate staleness of data. The following points indicate the reasons for such low errors:

- Even if the overall cache line is invalidated, it does NOT mean that the actual data word being read is stale because of (a) False sharing, (b) Silent stores/Spinlocks, etc.
    1) FALSE SHARING: In fact, up to 37% of coherence misses [21] are due to false sharing, where a different word has been written to invalidate the cache line with no change (and no loss of accuracy) done to the data being read by the load miss.
    2) SPINLOCKS/SILENT STORES: These again do NOT compromise any accuracy to the load, since successive stores causing the invalidations only write the same data values.
- Only the remaining cases could potentially pose chances of erroneous outputs. However, even in many of those cases, the self-correcting nature of our applications (as given in Table II), significantly reduces the consequences of using stale data. For example, the following piece of code from benchmark "FMM", shows how input smoothening automatically compensates for some level of staleness.

```
 FMM/interactions.C:540:
/*Each thread compute new positions
for all particles in each iteration.
No matter what the correct pos.(x,y)
values are, if x_sep and y_sep are
within a range, the accuracy is preserved*/
   x_sep = b->particles[i]->pos.x -
  b->particles[j]->pos.x;
 y_sep = b->particles[i]->pos.y -
  b->particles[j]->pos.y;

 if ((fabs(x_sep) < Softening_Param)
    && (fabs(y_sep) < Softening_Param)) {
    if (x_sep >= 0.0)
       x_sep = Softening_Param;
    else
       x_sep = -Softening_Param;
    if (y_sep >= 0.0)
       y_sep = Softening_Param;
    else
       y_sep = -Softening_Param;
 }
```

It is also possible that by supplying stale values, we finish faster than the ideal by skipping some iterations/converge to an agreeable point. Note that, our benchmarks do not have a single right output. Also, the accuracy checking is not just a "diff" of the outputs; rather, the checkers measure the correctness of the work that is done by the application and a range of output values could still be acceptable. So, it is possible for an approximate execution to find one of the acceptable outputs in a time faster than the Ideal case. We will present the details of such execution in this technical report [42].

Energy Consumption:: The additional energy consumed is characterized by the % change in the instructions executed. In addition to this, we measured using CACTI [43] that an 8-line SVC consumes 0.025nJ per (read/write) access in the 32 core setup.

C. Sensitivity Studies

We have also conducted an extensive study across different parameters, and we briefly summarize those results. The accuracies across the design space are comparable to those already presented, and we mainly focus on performance in the interest of space.

1) Sensitivity to Number of Cores: We compare the performance gains from using RIL, SVC and SVC+TB(100) for 8, 16 and 32 core configurations in Figure 7 (right). The corresponding percentages of coherence misses served from d-L1 and SVC in these 3 mechanisms are in Figure 7 (left).

The percentage of coherence misses served by the d-L1 increases with the number of cores across the schemes. Since

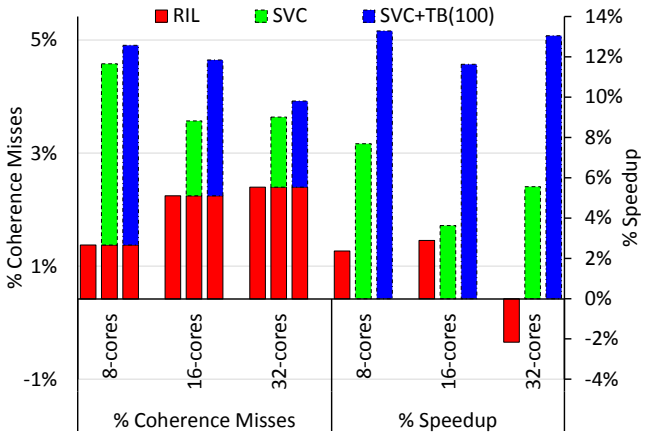
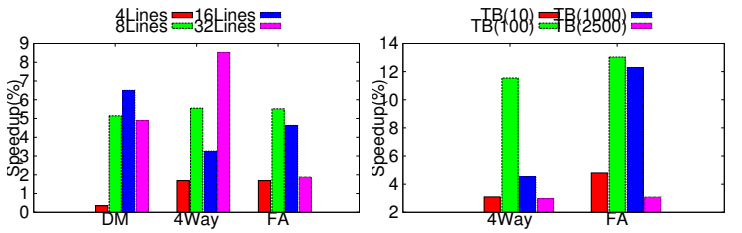


Fig. 7: Sensitivity to number of cores averaged across all benchmark. %Coherence Misses are shown on the Left y-axis and % Speedup on the Right y-axis.



(a) Varying capacity and associativity of SVC

(b) Varying TB and associativity of SVC+TB

Fig. 8: Sensitivity to SVC configurations averaged across all benchmarks.

working set size per core typically decreases as cores are increased for a fixed size problem, the capacity and conflict misses are likely to go down. This puts less pressure on space for invalidated lines in d-L1, giving more opportunity to serve stale values. However, as invalidations increase with the number of cores, the number of invalidated lines will itself increase and put more pressure on the relatively small SVC structure. Consequently, both SVC and SVC+TB see decrease in stale values served. RIL is not able to take advantage of the higher space offered in the existing d-L1, since the staleness value also increases for those lines with increasing cores. The speedup goes from 2.36% for 8 cores, to 2.89% for 16 cores, and reverses direction for a slowdown of -2.15% for 32 cores. Beyond this problem, SVC suffers from not being able to maintain the right set of lines within the victim cache without becoming too stale and hence the benefits are not consistent across the cores (speedups of 7.7%, 3.6% and 5.5% for 8, 16 and 32 cores) despite doing much better than RIL. SVC+TB, on the other hand, is able to better utilize its limited small capacity in the victim cache without letting the lines get too stale, and provides good consistent performance across the 3 core combinations (speedups of 13.3%, 11.6% and 13.03% for 8, 16 and 32 cores).

2) Sensitivity to SVC configurations: Figure 8a compares the average speedup obtained with SVC for varying number of lines (size) and associativities (Direct-Mapped, 4-way and Fully Associative). As can be seen, a 4-line DM SVC is too small to handle potential conflicts of stale lines, and is not able to provide much speedup. While higher associativities help somewhat, the speedups are still quite limited. Going to a capacity of 8 lines and beyond does help to lower these conflicts and serve more coherence misses. However, note that as SVC sizes get larger (and also at high associativities), stale lines will linger in the SVC for a long time, thereby growing in staleness. This detrimental effect can start hurting speedup, and we see evidence of this with higher SVC sizes and/or higher associativities. A 4-way SVC appears to be a good trade-off, and is the configuration used in earlier studies.

While SVC+TB can time-bound the staleness, and potentially benefit from a larger SVC, we found that a 8-line structure suffices and there is really not much benefit going for a larger size (not explicitly shown in the interest of space). Instead, we study different thresholds for the time-bound in Figure 8b, for a 8-line victim cache and associativities of 4-way and Fully Associative. A very small threshold could prematurely throw out stale lines, even before a load can benefit from them. On the other hand, a very high threshold could make the lines get very stale, making SVC+TB default to a SVC. As can be seen in the results, a time bound of 100 cycles is an appropriate trade-off point in these benchmarks, which is the value used in the previous experiments.

VI. Concluding Remarks and Future Work

We have explored the feasibility of leveraging approximations to serve coherence-induced load misses which can have an important consequence on the scalability of future CMPs. Specifically, we have examined the opportunistic use of stale versions of the lines to serve such load misses. Despite the promise, we find that using existing d-L1 space (in RIL) is not as fruitful, since those lines get evicted soon. On the other hand, using a very small Stale Victim Cache (SVC) to hold those lines, can double the number of such loads that can be served. However, without careful consideration of what data to keep and for how long, the SVC lines can become very stale. Such staleness can impact detrimentally as was demonstrated. To address these concerns, we propose a time bound for such lines in the SVC+TB scheme. The resulting mechanism provides as high as 28.6% speedup, with an average speedup between 10-15% across the SPLASH-2 benchmark suite. Despite this approximation, none of the applications crashed. Furthermore, there were no errors at all in the output in 11 of the benchmarks, and the maximum error that was observed was only 0.08%, clearly demonstrating the resiliency and/or tolerability of these applications to this approximation.

We have used a fairly simple approximation at the entire application granularity to determine whether to short-circuit the coherence-induced load misses with stale values. Given

the promise shown in this work, there is considerable scope for selectively performing such approximations - which applications? which data structures within the application? At what epochs in the execution? Programming, compiler and runtime support for addressing these considerations are part of our future work.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants 1213052, 1205618, 1302557, 1526750, 1409095, and 1439021, and a grant from Intel.

REFERENCES

[1] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching," in *Algorithmica*, 1988, pp. 79–119.

[2] F. Dahlgren and P. Stenstrom, "Reducing the write traffic for a hybrid cache protocol," in *Parallel Processing, 1994. Vol. 1. ICPP 1994. International Conference on*, 1994, pp. 166–173.

[3] H. Nilsson and P. Stenstrm, "An adaptive update-based cache coherence protocol for reduction of miss rate and traffic," in *Proc. Parallel Architectures and Languages Europe (PARLE) Conf., Athens, Greece (Lecture Notes in Computer Science, 817*. Springer-Verlag, 1994, pp. 363–374.

[4] S. V. Adve and M. D. Hill, "Weak ordering - a new definition," in *SIGARCH Comput. Archit. News*, 1990, pp. 2–14.

[5] M. Dubois, C. Scheurich, and F. A. Briggs, "Memory access buffering in multiprocessors." in *ISCA*, 1986, pp. 434–442.

[6] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *DAC*, 2013, pp. 1–9.

[7] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?" in *SIGARCH Comput. Archit. News*, 1999, pp. 162–171.

[8] B. R. Gaines, "Stochastic computing." in *AFIPS Spring Joint Computing Conference*, 1967, pp. 149–156.

[9] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance." in *HPCA*, 2007, pp. 181–192.

[10] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *SIGPLAN Not.*, 2011, pp. 164–174.

[11] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 13–24.

[12] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna, "Exploiting the forgiving nature of applications for scalable parallel execution," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010, pp. 1–12.

[13] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010, pp. 198–209.

[14] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation." in *SIGSOFT FSE*, 2011, pp. 124–134.

[15] S. Misailovic, D. M. Roy, and M. C. Rinard, "Probabilistically accurate program transformations." in *SAS*, 2011, pp. 316–333.

[16] V. Chippa, A. Raghunathan, K. Roy, and S. Chakradhar, "Dynamic effort scaling: Managing the quality-efficiency tradeoff," in *DAC*, 2011, pp. 603–608.

[17] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: an architectural framework for software recovery of hardware faults." in *ISCA*, 2010, pp. 497–508.

[18] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *SIGPLAN Not.*, 2011, pp. 199–212.

[19] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *SIGPLAN Not.*, 2012, pp. 301–312.

[20] J. S. Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in *MICRO*, 2014, pp. 127–139.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ISCA*, 1995, pp. 24–36.

[22] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *SIGARCH Comput. Archit. News*, 1990, pp. 364–373.

[23] "Amd x86-64 architecture programmer's manual. volume 2: System programming," in *AMD 64 Bit Technology*, 2002, pp. 197–214.

[24] A. Ros, M. E. Acacio, and J. M. Garcia, "A direct coherence protocol for many-core chip multiprocessors." in *IEEE Trans. Parallel Distrib. Syst.*, 2010, pp. 1779–1792.

[25] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: Decoupling performance and correctness." in *ISCA*, 2003, pp. 182–193.

[26] J. Owen, M. Hummel, D. Meyer, and J. Keller, "System and method of maintaining coherency in a distributed communication system," 2006, uS Patent 7,069,361.

[27] A. Raynaud, Z. Zhang, and J. Torrellas, "Distance-adaptive update protocols for scalable shared-memory multiprocessors," in *High-Performance Computer Architecture, 1996. Proceedings., Second International Symposium on*, 1996, pp. 323–334.

[28] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The stanford dash multiprocessor," in *Computer*, 1992.

[29] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors," in *Journal of Parallel and Distributed Computing*, 1991, pp. 87–106.

[30] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating prefetcher-caused pollution using informed caching policies for prefetched blocks," in *ACM Trans. Archit. Code Optim.*, 2015, pp. 51:1–51:22.

[31] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *ACM SIGARCH Computer Architecture News*, 1990, pp. 15–26.

[32] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence decoupling: making use of incoherence." in *ASPLOS*, 2004, pp. 97–106.

[33] P. Li and D. Lilja, "Using stochastic computing to implement digital image processing algorithms," in *ICCD*, 2011, pp. 154–161.

[34] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, "Randomized accuracy-aware program transformations for efficient approximate computations," in *SIGPLAN Not.*, 2012, pp. 441–454.

[35] B. Akgul, L. Chakrapani, P. Korkmaz, and K. Palem, "Probabilistic cmos technology: A survey and future directions," in *Very Large Scale Integration, 2006 IFIP International Conference on*, 2006, pp. 1–6.

[36] P. Bailis, S. Venkataraman, M. J. Franklin, J. M. Hellerstein, and I. Stoica, "Probabilistically bounded staleness for practical partial quorums," in *Proc. VLDB Endow.*, 2012, pp. 776–787.

[37] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing, "Solving the straggler problem with bounded staleness," in *HotOS*, 2013, pp. 22–22.

[38] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. R. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server." in *NIPS*, 2013, pp. 1223–1231.

[39] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, "Exploiting bounded staleness to speed up big data analytics," in *USENIX ATC*, 2014, pp. 37–48.

[40] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," in *SIGARCH Comput. Archit. News*, 2011, pp. 1–7.

[41] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Elsevier Science, 2011.

[42] P. V. Rengasamy, A. Sivasubramaniam, M. Kandemir, and C. Das, "Technical report cse-15-005, pennsylvania state university," Tech. Rep., 2015.

[43] P. Shivakumar, N. P. Jouppi, and P. Shivakumar, "Cacti 3.0: An integrated cache timing, power, and area model," HP-DEC, Tech. Rep., 2001.