On Approximate Speculative Lock Elision

S. Karen Khatamifard*  Ismail Akturk†  Ulya R. Karpuzcu*
* University of Minnesota  † University of Missouri
{khatami,ukarpuzc}@umn.edu  akturki@missouri.edu

Abstract—Each synchronization point represents a point of serialization, and thereby can easily hurt parallel scalability. As demonstrated by recent studies, approximating, i.e., relaxing synchronization by eliminating a subset of synchronization points spatio-temporally can help improve parallel scalability, as long as approximation incurred violations of basic execution semantics remain predictable and controllable. Even if the divergence from fully-synchronized execution renders lower computation accuracy rather than catastrophic program termination, for approximation to be viable, the accuracy loss must be bounded. In this paper, we assess the viability of approximate synchronization using Speculative Lock Elision (SLE), which was adopted by hardware transactional memory implementations from industry, as a baseline for comparison. Specifically, we investigate the efficacy of exploiting semantic and temporal characteristics of critical sections in preventing excessive loss in computation accuracy, and devise a light-weight, proof-of-concept Approximate Speculative Lock Elision (ASLE) implementation, which exploits existing hardware support for SLE.

Index Terms—Approximate Computing, mutual exclusion, parallel scalability.

1 INTRODUCTION

A simple type of synchronization, mutual exclusion, is crucial for the correct execution of parallel programs. Mutual exclusion restricts accesses (which involve updates) to shared data objects to one parallel task at a time. Lock-based synchronization serves this purpose. Before updating shared data objects, each parallel task has to first lock the critical section, where the respective data objects reside, to prevent simultaneous update attempts from other tasks. A typical parallel program may incorporate multiple critical sections protected by locks, which can be interleaved with each other in different ways. Independent of the frequency of occurrence or interleaving, each lock imposes a total or partial order on the execution of parallel tasks. Ergo, each lock represents a point of serialization, and thereby can easily hurt the scalability of parallel programs.

To enhance parallel scalability in the face of inevitable mutual exclusion, recent studies [29], [25], [28], [15] proposed to approximate mutual exclusion by eliminating a subset of locks spatio-temporally. The idea is to exploit the inherent noise tolerance of the emerging R(ecoognition), M(ining), and S(ythesis) applications [8] in mitigating approximation incurred violations of basic parallel execution semantics. RMS algorithms are iterative and often probabilistic to process massive yet noisy data. The solution space usually features a range of valid outputs [9]. Therefore, RMS applications can tolerate inaccuracies emanating from the data flow, as opposed to the control [37], [20], [36], [10]. In other words, RMS applications can mask approximation incurred semantic violations only if the divergence from fully-synchronized execution manifests as inaccuracies in the data flow. Even if approximation does not result in catastrophic program termination, for approximate mutual exclusion to be viable, the accuracy loss must remain bounded.

State-of-the-art optimizations for mutual exclusion focus on elimination of redundant, i.e., unnecessary, synchronization events—thereby, serialization points—without compromising computation accuracy [16], [27], [21], [3], [2]. Approximate mutual exclusion can complement these techniques by eliminating more synchronization points (in addition to the redundant) as long as the approximation incurred loss in computation accuracy remains at acceptable levels.

Approximation can apply to classic mutual exclusion implemented by locks, and as a more scalable alternative, to speculative (lock-based) synchronization such as Transactional Memory (TM) or Speculative Lock Elision (SLE) [16], [34], [14], [21], [27]. In this paper, we assess the viability of approximate mutual exclusion by approximating SLE, which was adopted by hardware transactional memory (HTM) implementations from industry [39]. In the following, we will refer to this HTM based baseline for comparison as SLE in short. Under SLE, parallel tasks do not attempt to lock critical sections before updates to shared data. Instead, they directly enter the critical section by speculating that no other task would attempt a simultaneous update. While correct speculation can eliminate serialization due to lock-based synchronization, misspeculation can result in conflicting accesses to shared data, which in turn can corrupt data values. SLE has to carefully track potential misspeculation to be able orchestrate recovery upon detection of misspeculation. To this end, all updates by speculative accesses should be buffered, and only reflected to the architectural state (i.e., committed) once speculation is deemed correct.

Approximate Speculative Lock Elision (ASLE), the focus of this study, translates into spatio-temporal omission of conflict detection and/or recovery upon misspeculation—only if potential loss in computation accuracy, as induced by potential data corruption due to conflicting accesses, remains acceptable. There is no need to buffer speculative data in this case as there is no need to recover. As a result, ASLE can cut-off the overhead incurred by conflict detection, speculative storage, and recovery (upon misspeculation).

In the following, by exploring the trade-off space of computation accuracy versus approximation-enabled speed-up, we assess the feasibility of Approximate spatio-temporal Speculative Lock Elision, ASLE. In accordance with recent work [29], [28],
2.1 Synchronization as a Barrier to Parallel Scalability

Accordingly, all threads, including the slowest, finish earlier, and the overall execution time reduces. At the same time, with increasing thread count, the number of sharers for a given chunk size, per thread work reduces as the thread count increases.

We observe that the synchronization overhead increases with higher degrees of parallelism. This is because, under a fixed problem size, per thread work reduces as the thread count increases. Accordingly, all threads, including the slowest, finish earlier, and the overall execution time reduces. At the same time, with increasing thread count, the number of sharers for a given chunk of data tends to grow, giving rise to more frequent synchronization.

For a representative set of RMS applications, Figure 1 captures how the time overhead of synchronization evolves as the number of threads (as a measure of the degree of parallelism) increases. The y-axis depicts, for each thread count, the percentage of the total execution time spent in synchronization events. Per Amdahl’s Law, time spent in synchronization increases from 4.4% to 48.8%; from 0.2% to 16.5%; and from 10.2% to 74.1% for ssca2, fluidanimate, and histo, respectively. This observation holds under an increasing problem size (aka weak scaling), as well. In this case, the work per thread remains constant where the thread count increases, which oftentimes results in more sharers, therefore, more frequent synchronization.

2.2 Approximate Mutual Exclusion: Challenges

Under approximate mutual exclusion, multiple independent accesses to modify shared data can proceed simultaneously. Consequently, approximation may not only corrupt data values residing in critical sections, but also prevent program termination, by e.g., giving rise to deadlocks. In mitigating approximation induced violations of basic execution semantics, the inherent noise tolerance of RMS applications can only help if the violations do not escape to control-flow, and do not result in excessive degradation in computation accuracy by corrupting data-flow [37], [20], [36], [10].

Under approximate mutual exclusion, even in the absence of catastrophic program termination, the magnitude of data corruption becomes hard to predict. Worse, approximation induced data value corruption can propagate to application outputs in numerous ways [19], [30]. The magnitude of corruption at the outputs depends on the sensitivity of program outputs to the corrupted data values (residing in critical sections subject to approximate mutual exclusion). During execution, this sensitivity may change temporally, as well. Most iterative RMS algorithms progressively refine their outputs each iteration, until meeting predefined convergence criteria [7]. Accordingly, iterations further in the execution may tolerate approximate mutual exclusion less than iterations at the beginning. Therefore, approximation may also affect how fast the algorithm converges to a solution.

2.3 Case Study: kmeans

A heavily used RMS application for data mining, kmeans, relies on iterative refinement (in its most standard form) in partitioning its input data points into k clusters. Every iteration enforces the assignment of each input point to the closest possible cluster. The algorithm terminates once assignments stabilize. kmeans from the STAMP benchmark suite [22] incorporates three critical sections:

The first protects the assignment of new points to clusters:

```
1 *new_centers_len[index] += *new_centers_len[index] + 1;
2 <Insert new point to list>
```

new_centers_len[index] keeps the number of points assigned to cluster index. Line 2 inserts the new point to the list containing all points assigned to cluster index. Under approximation, multiple threads may try to assign a point to cluster index simultaneously. This may corrupt the update to new_centers_len[index] in line 1, or the corresponding insertion of the point to the cluster’s list in line 2. A corruption in line 1 can easily render a lower (than actual) number of points per cluster. A corruption in line 2, on the other hand, may skew cluster centers. kmeans can mask both types of corruption due to the iterative refinement of cluster centers until convergence.

A heavily used RMS application for data mining, kmeans, relies on iterative refinement (in its most standard form) in partitioning its input data points into k clusters. Every iteration enforces the assignment of each input point to the closest possible cluster. The algorithm terminates once assignments stabilize. kmeans from the STAMP benchmark suite [22] incorporates three critical sections:

The first protects the assignment of new points to clusters:

```
1 *new_centers_len[index] += *new_centers_len[index] + 1;
2 <Insert new point to list>
```

new_centers_len[index] keeps the number of points assigned to cluster index. Line 2 inserts the new point to the list containing all points assigned to cluster index. Under approximation, multiple threads may try to assign a point to cluster index simultaneously. This may corrupt the update to new_centers_len[index] in line 1, or the corresponding insertion of the point to the cluster’s list in line 2. A corruption in line 1 can easily render a lower (than actual) number of points per cluster. A corruption in line 2, on the other hand, may skew cluster centers. kmeans can mask both types of corruption due to the iterative refinement of cluster centers until convergence.

The next critical section controls the distribution of input data points among threads:

```
start = global_i;
global_i = start + CHUNK;
```

1 We deploy the largest available input data set for each benchmark. Section 4 provides the experimental setup.
In this critical section, each thread tries to get CHUNK number of points for processing. The points reside in an array with global_i pointing to the index of the last point already assigned to a(nother) thread. Accordingly, the next thread in the row increments global_i by CHUNK. This critical section mainly affects control flow. Under approximation, multiple threads may end up processing the very same points. Such redundant processing may easily increase the overall execution time particularly if the thread count is no greater than the total number of chunks (i.e., number of sets of points of size CHUNK). At the same time, the accuracy may degrade as adding the same point to a cluster multiple times may skew cluster centers.

The final critical section protects convergence control:

\[
global\_delta = global\_delta + delta; \\
global\_delta \text{ captures the total number of new cluster assignments overall; } delta, \text{ in a given iteration. The algorithm terminates if } global\_delta \text{ falls below a predefined threshold. This critical section mainly affects control flow. Under approximation, multiple threads may try to update } global\_delta \text{ simultaneously. As a result, } global\_delta \text{ can possibly assume a lower value than under fully-synchronized execution, and trigger premature termination. Premature termination is likely to boost performance due to faster, yet false, convergence. The corresponding accuracy loss, however, may become excessive. Adjusting the threshold or imposing a fixed number of iterations may work better than approximate mutual exclusion in this case, as we will demonstrate in Section 5.}
\]

3 Approximate Speculative Lock Elision

We will next analyze the viability of approximate synchronization using Speculative Lock Elision (SLE), which was adopted by hardware transactional memory implementations from industry [39], as a baseline for comparison. In the following, we will refer to this HTM based baseline for comparison as SLE in short. The key difference of Approximate Speculative Lock Elision (ASLE) from SLE is spatio-temporal omission of conflict detection and/or recovery upon mis speculation – however, only if potential loss in computation accuracy, as induced by potential data corruption due to conflicting accesses, remains acceptable.

A key design question for ASLE hence becomes how to decide where in program and when to turn off conflict detection and/or recovery upon mis speculation – which is the equivalent of how to select the locks to elide in approximating classic mutual exclusion. The intuitive answer to this question is when the execution reaches an inaccuracy-tolerant critical section. Recall that the tolerance of the target application domain, RMS, to inaccuracy in computation comes from (i) mostly probabilistic algorithms often using iterative refinement; and (ii) inputs containing a very large number of inaccurate, often redundant data elements. Due to (i), it is barely possible to differentiate inaccuracy-tolerant critical sections from others without analyzing program semantics. Unfortunately, (ii) complicates the identification of inaccuracy-tolerant critical sections further, as the inaccuracy-tolerance tends to change as a function of inputs. Therefore, profiling-based identification of inaccuracy-tolerant critical sections becomes inevitable. These constraints do complicate programming. The proof-of-concept ASLE implementation covered in this study relies on profiling-based identification of inaccuracy-tolerant critical sections, similar to [23], [30], [25].

After identifying inaccuracy-tolerant critical sections, the next design question becomes how to turn off conflict detection and/or recovery upon mis speculation within this subset of critical sections – which is the equivalent of how to elide the locks in approximating classic mutual exclusion. A critical section may lend itself well to approximation, however, the corresponding accuracy loss may fluctuate at runtime due to variations in the degree of parallelism, size and characteristics of input data, or environmental conditions. Therefore, enforcing approximation at compile time may not always be safe. Instead, by communicating potentially inaccuracy-tolerant critical sections subject to approximation to the hardware, we can devise adaptive policies to control the degree of approximation at runtime. To this end, we can adapt programming language or instruction set extensions as suggested by [12], [32], [31], [28]. To demarcate potentially inaccuracy-tolerant critical sections subject to approximation, similar to AMD’s ASF [11] or Intel’s TSX [39] for SLE, the proof-of-concept ASLE implementation relies on compiler-managed instruction set extensions APPROX_ACQUIRE and APPROX_RELEASE.

We will next discuss the remaining open questions of how to control the degree of approximation and how to prevent excessive accuracy loss under approximation.

3.1 Knobs to Control the Degree of Approximation

ASLE can degrade computation accuracy only if multiple parallel tasks attempt to enter a critical section simultaneously, i.e., if accesses to shared data conflict. The magnitude of accuracy loss is likely to grow with an increasing number of conflicting accesses. In other words, approximation in higher contention critical sections is likely to render higher loss in computation accuracy as observed by previous work [15]. Based on this insight, we will next investigate how contention profiles can serve as a knob to adjust the accuracy loss due to approximation. In the following, we will refer to the number of conflicting accesses as a proxy for contention. On the other hand, the share of conflicting accesses over all accesses constitutes the conflict rate.

To quantify how strongly correlated contention and accuracy loss are, we experiment\(^2\) with a very aggressive type of approximate (speculative) mutual exclusion, where we never check for conflicts – and hence, (can) never trigger recovery. In the rest of the paper, we will refer to this as Brute-force Lock Elision (BLE). BLE is more likely to result in conflicts than ASLE. Under classic mutual exclusion, BLE corresponds to permanent spatio-temporal elision of the lock for the duration of the entire program. For this analysis, we only consider inaccuracy-tolerant critical sections (i.e., critical sections lending themselves well to approximation), as identified by profiling. For the representative RMS benchmarks deployed in this study we identify three distinctive cases:

i. Persistently low contention critical sections where the conflict rate assumes a very low value throughout execution, which does not change with contention.

ii. Relatively low contention critical sections where accuracy loss is strongly correlated (i.e., changes almost linearly) with conflict rate. Generally, conflict rate increases with contention, and a higher conflict rate renders higher accuracy loss.

iii. High contention critical sections where accuracy loss is strongly correlated (i.e., changes almost linearly) with conflict

\(^2\)We deploy the largest available input data sets and 64 threads. Section 4 details the experimental setup.
rate. Generally, conflict rate increases with contention, and a higher conflict rate renders higher accuracy loss.

We conclude that, modulo i., contention or conflict rate can serve as a practical knob to control the accuracy loss under approximation by ASLE. For i., BLE is a better option than ASLE, as we will demonstrate in Section 5.

3.2 A Proof-of-Concept ASLE Implementation

![Diagram](https://via.placeholder.com/150)

**Fig. 2:** A light-weight approximation extension (depicted in gray) to speculative lock elision (SLE).

Speculative Lock Elision (SLE) has to carefully track potential conflicts to be able orchestrate recovery upon conflict detection. The baseline SLE implementation already features Conflict Detection Logic which we can exploit to keep track of contention in controlling the degree of approximation. Figure 2 details a light-weight approximation extension (shaded in gray) to SLE in hardware. Under SLE (which does not feature any of the shaded approximation extensions) Conflict Detection Logic asserts the Conflict signal upon detection of conflicting accesses. Assertion of Conflict triggers the Retry signal in order to have conflicting accesses to shared data re-attempted.

Approximate SLE, ASLE, on the other hand, can opportunistically omit these Retry attempts, as long as degradation in computation accuracy due to conflicts remains at acceptable levels. The proof-of-concept ASLE implementation uses the very same Conflict Detection Logic as SLE to detect conflicts. However, under approximation, assertion of Conflict triggers the Retry signal only if the conflicting accesses can lead to excessive loss in computation accuracy. To control the degree of accuracy loss, i.e., approximation, ASLE relies on two buffers: Retry Threshold and Conflict History. Retry Threshold and Conflict History represent special purpose programmable registers per core. Retry Threshold controls the frequency of approximation, while Conflict History keeps track of contention.

Conflict History is an N-bit shift register, and stores the N most recent values of the Conflict signal. Under ASLE, each thread attempts to update the architectural state once done with the execution of a critical section subject to approximation. At this stage, if there was no conflict, i.e., the thread finds the Conflict signal not set, no retry is necessary (i.e., Retry signal is not set). Otherwise, if there was a conflict, i.e., the thread finds the Conflict signal set, Retry is only triggered if we disable approximation by resetting the Approximate signal from Figure 2. We next look into how the proof-of-concept ASLE implementation manages the Approximate signal.

3.3 Runtime Policies to Control Accuracy Loss

We set Approximate from Figure 2 as a function func of how the number of conflicts (encountered during the N most recent commit attempts, on a per core basis) compares to a given threshold, which is stored in Retry Threshold. The number of conflicts encountered during the N most recent commit attempts corresponds to the number of i’s in Conflict History. We next devise several runtime policies (each featuring a different func) which span the trade-off space of accuracy loss vs. speed-up under approximation.

Under the first policy, we trigger Approximate if the number of conflicts (encountered during the N most recent commit attempts, on a per core basis) is Higher-Than Retry Threshold. At the end of execution of each critical section subject to approximation,

- if Conflict = 0: ASLE falls back to SLE (Case-1).
- if Conflict = 1 and Conflict History keeps a conflict count lower than Retry Threshold: ASLE falls back to SLE (Case-2).
- if Conflict = 1 and Conflict History keeps a conflict count higher than Retry Threshold: by setting Approximate (which in turn resets Retry), we let conflicting tasks proceed without attempting retry (Case-3).

Higher-Than thereby enables approximation only if contention, i.e., the number of conflicting accesses, exceeds a preset threshold. Therefore, **Higher-Than is biased to approximate higher contention critical sections, which render a higher number of conflicts. Approximating predominantly higher contention critical sections may deliver higher speed-up (as higher contention implies more retries which ASLE can effectively cut-off), however, the accompanied loss in accuracy may become hard to bound. Under Higher-Than, a higher value of Retry Threshold leads to less frequent approximation, and therefore, (more likely) to less accuracy loss.**

We define the dual policy of Higher-Than as Lower-Than, by inverting the sense of the threshold logic for Case-2 and Case-3. Under Lower-Than,

- if Conflict = 0: ASLE falls back to SLE.
- if Conflict = 1 and Conflict History keeps a conflict count higher than Retry Threshold: ASLE falls back to SLE.
- if Conflict = 1 and Conflict History keeps a conflict count lower than Retry Threshold: by setting Approximate (which in turn resets Retry), we let conflicting tasks proceed without attempting retry.

Contrary to Higher-Than, Lower-Than is biased to approximate lower contention critical sections, which render a lower number of conflicts. Approximating predominantly lower contention critical sections may deliver modest speed-up, however, the accompanied loss in accuracy is likely to be modest, as well. Under Lower-Than, a higher value of Retry Threshold leads to more frequent approximation, and therefore, (more likely) to more accuracy loss. **Putting It All Together:** Under Higher-Than, ASLE reacts to high contention, and triggers approximation mainly to boost performance. Under Lower-Than, ASLE reacts to low contention, and triggers approximation mainly to keep the accuracy loss bounded, which is accompanied by a more modest speed-up than Higher-Than. Under both policies, Retry Threshold controls the
frequency of approximation, therefore, the loss in accuracy. Fine-tuning the value of Retry Threshold, possibly dynamically at runtime, can help prevent excessive loss in computation accuracy.

An application can feature many critical sections, each exhibiting different contention characteristics. Figure 2 shows basic hardware support to keep track of a single critical section. However, if critical sections of an application are sufficiently apart from each other in time, this hardware can also accommodate accurate support for multiple critical sections. Otherwise, we can introduce an array of Conflict History along with an array of Retry Threshold, each array element to keep track of a separate critical section. RMS applications we experiment with do not feature more than a few (static) inaccuracy-tolerant critical sections, as Section 5 reveals.

3.4 Practical Limitations

Demand for application specific characterization: Not every critical section lends itself well to approximation as demonstrated in Section 2.3. Oftentimes, selection of locks subject to elision demands significant semantic knowledge about the application. Profiling may help, but cannot cover all possible use cases. For ASLE policies, we relied on profiling, first, to identify critical sections which (when relaxed) do not lead to catastrophic termination, in the form of non-termination or excessive degradation in output accuracy. We excluded these critical sections from approximation. For the rest, we identified, through a more detailed profiling step, higher contention critical sections and adjusted the ASLE thresholds accordingly, based on the average contention rates.

Confining approximation-induced inaccuracies in data-flow: Exploiting contention profiles, ASLE policies can only qualitatively control the accuracy loss. While ASLE policies (particularly, the Lower-Than variants) can probabilistically prevent excessive accuracy loss, they fail short of bounding its magnitude. To be able to bound the magnitude of accuracy loss, we need to quantitatively characterize how approximation-induced inaccuracies in the data flow propagate to the application output to degrade the output accuracy. An important step in addressing this fundamental limitation is adapting previous work such as [10], [17] to enforce approximation incurred inaccuracies confined in data flow where RMS applications can tolerate corruption, as opposed to control flow. Decoupling data flow from control is already a challenging task beyond the scope of this paper.

Need for safety-nets: Even if we effectively managed to confine approximation-induced inaccuracies to data flow of RMS applications, there is no deterministic guarantee that data flow errors do not result in catastrophic program termination. Accordingly, for ASLE to be viable, we still need to devise safety-nets (e.g., based on checkpoint-recovery) to facilitate recovery without compromising design complexity. A more viable solution is enabling approximation only under very strong statistical guarantees.

4 Evaluation Setup

Benchmarks: As captured by Table 1, we deploy a representative set of RMS applications from PARSEC [5], Parboil [35], STAMP [22], SPLASH2x [38], and RMS-TM [18] suites. barnes and fluidanimate represent n-body simulations. histo computes a large, 2-D saturating histogram. kmeans implements an iterative clustering algorithm. ssc2 consists of four graph kernels; we use the first kernel which constructs an efficient graph data structure. tpacf generates a histogram capturing the spatial distribution of astronomical observations. utilitymine, an Associate Rule Mining (ARM) application, extracts high-utility itemsets from a database.

Approximation Targets: We confine approximation to critical sections (CS) protected by locks or transactions. The applications feature critical sections of different length, (dynamic execution) frequency, and contention characteristics, as depicted in Table 2. We focus on critical sections exercised frequently enough to qualify as potential performance bottlenecks. Further, if approximation of a critical section is likely to result in catastrophic program termination, we exclude it from consideration.

Metrics to Quantify Accuracy Loss: To quantify the accuracy loss due to approximation, we adapt accuracy metrics from Misiailovic et al. [24] along with Akturk et al. [1]. The last column of Table 1 depicts the accuracy metrics.

Simulation Infrastructure: We deploy Sniper [6] for microarchitectural simulation, as configured in Table 3. We implement all policies from Section 3 in Sniper. The baseline SLE implementation represents a TSX-like design [39], which we evaluate under both eager and lazy conflict detection. Not to compromise simulation speed, Sniper tends to model the interactions among threads at a coarse-granularity, which may lead to implicit serialization. A similar restriction applies to many microarchitectural simulators. To mitigate this effect, we increase the simulator’s thread synchronization frequency to its maximum value. To establish statistical significance of our results, we repeat each experiment 100 times, and report the mean. To capture the sensitivity of the execution outcome to inputs, we use a range of input data-sets for each application, as tabulated in the third column of Table 1. Our simulations involve up to 64 threads. For Parboil applications we implemented the pthread versions, and made sure that the pthread implementation outperformed the omp-based baseline not to favor any ASLE policy.

5 Evaluation

5.1 Impact on Execution Time

We first compare and contrast the execution time under classic locking (Classic) with speculative lock elision (SLE) (implemented in hardware), and with persistent, spatio-temporal brute-force lock elision (BLE), for each application. We deploy the largest input data set for each benchmark. For each lock implementation, we report parallel efficiency as the ratio of the speed-up (over the single-threaded execution) to the maximum possible speed-up for a given thread count. For example, if an 64 threaded run of a benchmark delivers 50× speed-up over single-threaded execution, parallel efficiency becomes 50/64 ≈ 78%. In Figure 3, the y-axis provides the % parallel efficiency, where each bar corresponds to a benchmark, and each stack depicts the % improvement in parallel efficiency under a particular lock implementation.

We observe that lock elision (be it BLE or SLE) does not improve the efficiency of barnes. For fluidanimate, BLE improves the efficiency by 8.8% over SLE, which itself delivers 2.3% more efficiency over Classic. These efficiency numbers evolve to 49.0% and 51.1% for histo; to 29.7% and 39.81% for ssc2; and to 18.2% and 53.8% for tpacf. Finally, utilitymine experiences 77.5% efficiency under SLE, and an additional 11.3%, under BLE for 8-threads (maximum number of threads for this application). For kmeans, we restrict the comparison to a single iteration of the application (as explained in Section 5.4). In this case, SLE delivers 21.5%; BLE, 52.9% efficiency.
TABLE 1: RMS benchmarks deployed.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Input files</th>
<th>Accuracy metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes (splash2x)</td>
<td>n-body Simulation</td>
<td>simsmall of splash2x simmedium of splash2x</td>
<td>avg. relative deviation of coordinates</td>
</tr>
<tr>
<td>fluidanimate (RMS-TM 3)</td>
<td>n-body Simulation</td>
<td>simsmall of parsec 2.0 simlarge of splash2x</td>
<td>avg. relative deviation of coordinates</td>
</tr>
<tr>
<td>histo (Parboil 2.5)</td>
<td>Saturating Histogram</td>
<td>(-i) img.bin (-i) uniform.in (-i) gaussian.in</td>
<td>avg. relative deviation of bin values</td>
</tr>
<tr>
<td>kmeans (STAMP)</td>
<td>Data Mining</td>
<td>(-m5) -n15 -t0.001 -i random-n2K (-m5) -n15 -t0.0001 -i random-n16K (-m5) -n15 -t0.00001 -i random-n65K</td>
<td>ratio of wrong clusterings</td>
</tr>
<tr>
<td>ssc2a (STAMP)</td>
<td>Graph Analysis</td>
<td>(-s1) -l1.0 -ul1.0 -l1.0 -p3 (-s1) -l1.0 -ul1.0 -l1.0 -p3 (-s1) -l1.0 -ul1.0 -l1.0 -p3</td>
<td>ratio of differences in the number of edges</td>
</tr>
<tr>
<td>tpacf (Parboil 2.5)</td>
<td>Two Point Angular Correlation Function</td>
<td>(-n) 100 -p 487 (-n) 100 -p 4096 (-n) 100 -p 10391</td>
<td>avg. relative deviation of bin values</td>
</tr>
<tr>
<td>utilitymine (RMS-TM 3)</td>
<td>Association Rule Mining</td>
<td>data...nitems_10.pnglen_6 0.01 data...nitems_1.pnglen_6 0.01</td>
<td>avg. relative deviation of utilities</td>
</tr>
</tbody>
</table>

TABLE 2: Critical sections subject to approximation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>File(line)</th>
<th>Length</th>
<th>Frequency</th>
<th>Cont</th>
<th>Protection</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes</td>
<td>code.C(721)</td>
<td>Long</td>
<td>Medium</td>
<td>Low</td>
<td>global min and max</td>
</tr>
<tr>
<td>fluidanimate</td>
<td>pthreads.cpp(635)</td>
<td>Short</td>
<td>High</td>
<td>Low</td>
<td>cell density</td>
</tr>
<tr>
<td></td>
<td>pthreads.cpp(641)</td>
<td>Short</td>
<td>High</td>
<td>Low</td>
<td>cell density</td>
</tr>
<tr>
<td></td>
<td>pthreads.cpp(744)</td>
<td>Short</td>
<td>High</td>
<td>Low</td>
<td>cell acceleration</td>
</tr>
<tr>
<td></td>
<td>pthreads.cpp(761)</td>
<td>Short</td>
<td>High</td>
<td>Low</td>
<td>cell acceleration</td>
</tr>
<tr>
<td>histo</td>
<td>main.c(1003)</td>
<td>Short</td>
<td>High</td>
<td>Low</td>
<td>bins update</td>
</tr>
<tr>
<td>kmeans</td>
<td>normal.c(108)</td>
<td>Long</td>
<td>High</td>
<td>High</td>
<td>center update</td>
</tr>
<tr>
<td>ssc2a</td>
<td>c...Graph.cpp(45/5)</td>
<td>Long</td>
<td>High</td>
<td>High</td>
<td>add edge to list</td>
</tr>
<tr>
<td>tpacf</td>
<td>model_common.c(52)</td>
<td>Short</td>
<td>High</td>
<td>High</td>
<td>bins update</td>
</tr>
<tr>
<td>utilitymine</td>
<td>utility.cpp(201)</td>
<td>Short</td>
<td>Medium</td>
<td>Low</td>
<td>update utility</td>
</tr>
</tbody>
</table>

TABLE 3: Architectural parameters.

<table>
<thead>
<tr>
<th># of cores</th>
<th>#33</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>4-issue wide 640</td>
</tr>
<tr>
<td>Private L1</td>
<td>32KB 8-way, 64B</td>
</tr>
<tr>
<td>Private L1</td>
<td>12KB 8-way, 64B</td>
</tr>
<tr>
<td>Private L2</td>
<td>32KB 8-way, 64B</td>
</tr>
<tr>
<td>Shared L3</td>
<td>120MB 16-way, 64B</td>
</tr>
<tr>
<td>Main memory</td>
<td>384MB 16-way, 64B</td>
</tr>
<tr>
<td>Technology</td>
<td>2.2GHz</td>
</tr>
<tr>
<td>Voltage/ frequency</td>
<td>1.0V/1.05GHz</td>
</tr>
</tbody>
</table>

![Fig. 4: Speed-up and concurrency profile for barnes (a,b) and ssc2a (c,d).](image)

![Fig. 3: Parallel efficiency under lock elision.](image)

Figures 4a and 4b provide the corresponding speed-up and concurrency profiles for barnes. The speed-up is normalized to the execution time of the single-threaded run under Classic, and reported as a function of the thread count on the x-axis. From Figure 4a, we observe that lock elision does not deliver any speed-up for this application. The concurrency profile from Figure 4b – which captures the % of execution time (as depicted on the y-axis) the application features a specific number of concurrently active threads (as depicted on the x-axis) – verifies this trend. Figures 4c and 4d provide the same analysis for ssc2a, which, as opposed to barnes, demonstrates enhanced concurrency under lock elision.

5.2 Impact on Accuracy Loss

Figure 5 captures how BLE changes the computation accuracy to accompany the (potential) efficiency improvement from Figure 3. Classic and SLE do not compromise the computation accuracy. The analysis reflects 64-threaded runs for the largest input set. The figures report the distribution of % accuracy loss over 100 runs. No accuracy loss applies to barnes, since the execution trajectory under BLE closely follows the execution trajectory under Classic or SLE. fluidanimate (Figure 5a) shows a modest accuracy loss –
less than 0.092% – due to the very low contention of its locks. The accuracy loss remains negligible for histo, as well (Figure 5b). This is because histo features a very large number of bins (in the order of hundreds), which decreases the probability of the same bin being accessed by multiple threads, even under high contention. We will provide a detailed accuracy analysis for kmeans in Section 5.4. ssc2 exhibits an accuracy loss of 0.0031%, on average. tpac2 also has a histogram as its output, but suffers from a significantly higher accuracy loss than histo due to the lower number bins (in the order of 20) (Figure 5d). The % accuracy loss distribution under utility mine assumes a similar pattern to tpac2 over a range of [0.009 – 0.011]% – although the utilities in the output of utility mine degrade slightly, the final high-utility itemsets very closely follow the exact outcome, i.e., the outcome under Classic or SLE. Overall, the modest % accuracy loss observed across all applications renders the approximation-enabled parallel efficiency enhancement under BLE viable (Figure 3).

5.3 Controlling Accuracy Loss

We next characterize how the trade-off space for % accuracy loss vs. % (parallel) efficiency (Section 5.1) evolves under policies from Section 3.3. We report % accuracy loss on the y-axis and % efficiency on the x-axis. All simulations reflect 64-threaded runs, using the largest input size. In the following, we only consider applications with non-negligible contention. Figure 6 shows the trade-off space under Higher-Than and Lower-Than policies, considering different values of Retry Threshold for tpac2. In this case, Conflict History keeps a very small number (of ones) throughout the execution. We adjust the range for Retry Threshold accordingly. For instance, Higher-Than 1 policy enforces approximation only if the one count of Conflict History is greater than 1, to render an \( \approx 2.9 \times \) more accurate result than under BLE, where efficiency falls by \( \approx 30\% \) beyond BLE \( (H > 1) \) from Figure 6a. Lower-Than 1 policy, on the other hand, renders an \( \approx 1.9 \times \) more accurate result than under BLE, where efficiency falls by more than 20% beyond BLE \( (H < 1) \) in Figure 6b.

Following our observations from Section 3.1, we experiment with larger values of Retry Threshold for the higher contention application ssc2. Figure 6 depicts the trade-off space. In this case, Higher-Than 45 \( (H > 45) \) policy improves the accuracy by 37.3% over BLE, accompanied by a more than 5% reduction in efficiency (Figure 6c). Lower-Than, \( H < 45 \) policy, on the other hand, improves the accuracy by 66.3% over BLE, accompanied by around 8% reduction in efficiency (Figure 6d).

Overall, we observe that this basic set of policies can span a rich trade-off space, which we can exploit to deliver the optimal accuracy loss vs. parallel scalability under varying application-specific constraints. The programmer or the runtime can choose between these two policies depending on the relative importance of speed-up vs. accuracy.

5.4 Case Study: kmeans

Since kmeans relies on iterative refinement, by enforcing a lower number of iterations until convergence than the fully-synchronized baseline execution, we can trade accuracy for speed-up, without approximating synchronization. To quantify this effect, we turned off the convergence check of the algorithm, and manually enforced a fixed number of iterations \( (#iter) \) for each run, under both SLE and BLE, for 64 threads. Figure 7 summarizes our findings: Each point corresponds to a fixed iteration count, \#iter (which increases as we move from top-right to bottom-left). The x-coordinate of each point captures the speed-up under \#iter; the y-coordinate, the corresponding % loss in accuracy. The speed-up is reported with respect to the baseline SLE without any approximation under the default iteration count until termination (i.e., by enforcing the default convergence check). The speed-up reduces with increasing values of \#iter, the number of iterations executed until termination. The triangle demarcates the outcome under BLE; the % accuracy loss on the y-axis, the speed-up on the x-axis. The red rectangle demarcates the region of “higher speed-up at lower accuracy loss” than possible under BLE. We observe that \#iter=2 indeed renders a point in this more favorable region than BLE. Accordingly, enforcing a fixed number of iterations may result in a better accuracy versus speed-up trade-off than possible by enforcing BLE.
5.5 Sensitivity Analysis

Figure 8 depicts how the distribution of % loss in accuracy evolves with an increasing thread count for the largest input of \( \text{tpacf} \) under \( \text{BLE} \). A higher number of threads leads to a higher number of conflicts, which progressively increases the % accuracy loss. A similar trend applies for all of the RMS benchmarks we experimented with.

![Fig. 8: Sensitivity of % accuracy loss to thread count.](image)

However, the picture changes across different benchmarks when we deploy different input sizes. We observe two distinct patterns, as captured in Figure 9 for two representative applications, \( \text{tpacf} \) (Figure 9b) and \( \text{ssca2} \) (Figure 9a), respectively. A larger input size renders a higher % accuracy loss for \( \text{ssca2} \), while the opposite trend applies for \( \text{tpacf} \).

![Fig. 9: Sensitivity of % accuracy loss to input size.](image)

So far, without loss of generality, for (A)SLE we assumed \( \text{Lazy} \) conflict detection; i.e., conflict detection being fired at the end of critical sections. \( \text{Eager} \) [26] conflict detection, on the other hand, would trigger the \text{Conflict} signal immediately upon identification of conflicts inside the critical section. The performance of applications under these two policies may vary. For instance, \( \text{histo} \) and \( \text{tpacf} \) show very similar parallel efficiency under \( \text{Eager} \) to \( \text{Lazy} \) (with less than 0.5% difference), since they both feature a very short critical section. For applications with longer critical sections, such as \( \text{ssca2} \), the difference becomes more visible: for example, parallel efficiency of \( \text{ssca2} \) improves from 29.7% under \( \text{Lazy} \) to 33.2% under \( \text{Eager} \) conflict detection.

![Fig. 10: Trade-off space for \( \text{tpacf} \) under \( \text{Eager} \) conflict detection.](image)

Figure 10 depicts how the trade-off space of accuracy loss versus parallel efficiency looks like under \( \text{Eager} \) conflict detection for \( \text{tpacf} \). Under \( \text{Eager} \), \( \text{ASLE} \) still delivers a rich trade-off space of accuracy vs. speed-up. A similar trade-off space applies for other applications, as well. Comparing \( \text{Eager} \) and \( \text{Lazy} \) conflict detection, only the range of parallel efficiency benefits varies.

6 RELATED WORK

Effective techniques to mitigate the overhead of synchronization [3], [2], [33], including speculation [16], [27], [21], [34], [14], eliminate unnecessary synchronization events without compromising accuracy. By eliminating even more synchronization points, approximate synchronization can complement these techniques [31], [25], [28], [15]. Under speculative synchronization, parallel tasks (threads or transactions) proceed speculatively past synchronization points. If speculation does not result in conflicting accesses to shared data or violation of parallel execution semantics, this class of techniques can eliminate serialization due to synchronization, at the cost of extra storage for speculative state, control logic to detect violations of parallel execution semantics, and to orchestrate roll-back to a safe state and retry in case of misspeculation. Approximation can mitigate (if not eliminate) the overhead of storage, conflict detection, or recovery incurred by speculation. While previous studies on approximate synchronization focus mostly on programming language implications or basic quantitative characterization [31], [25], [28], we explore a transparent hardware extension to speculative lock elision in order to orchestrate approximation.

7 DISCUSSION & FUTURE WORK

Under persistent high contention, due to a monotonic increase in conflict count (and independent of the value of \( \text{Retry} \).
improvement of up to 35% at negligible accuracy loss is possible. Otherwise, speculative techniques are more promising, since they do not compromise accuracy. A notable reduction in execution time due to approximation applies for the majority of the benchmarks. For the remaining, we may still expect higher speed-up with increasing degrees of parallelism per Amdahl’s Law.

References


