Approximate Lock: Trading off Accuracy for Performance by Skipping Critical Sections

Riad Akram, Mohammad Mejbah ul Alam, Abdullah Muzahid
University of Texas at San Antonio
{*riad.akram, mohammad.alam, abdullah.muzahid*}*@utsa.edu*

***Abstract*—Approximate computing is gaining a lot of traction due to its potential for improving performance and consequently, energy efficiency. This project explores the potential for approximating locks. We start out with the observation that many applications can tolerate occasional skipping of computations done inside a critical section protected by a lock. This means that for certain critical sections, when the enclosed computation is occasionally skipped, the application suffers from quality degradation in the final outcome but it never crashes/deadlocks. To exploit this opportunity, we propose Approximate Lock (ALock). The thread executing ALock checks if a certain condition (e.g., high contention, long waiting time) is met and if so, the thread returns without acquiring the lock. We modify some selected critical sections using ALock so that those sections are skipped when ALock returns without acquiring the lock. We experimented with 14 programs from PARSEC, SPLASH2, and STAMP benchmarks. We found a total of 37 locks that can be transformed into ALock. ALock provides performance improvement for 10 applications, ranging from 1.8% to 164.4%, with at least 80% accuracy.**

***Keywords*-Multithreaded program; Lock; Approximate computing; Accuracy; Performance;**

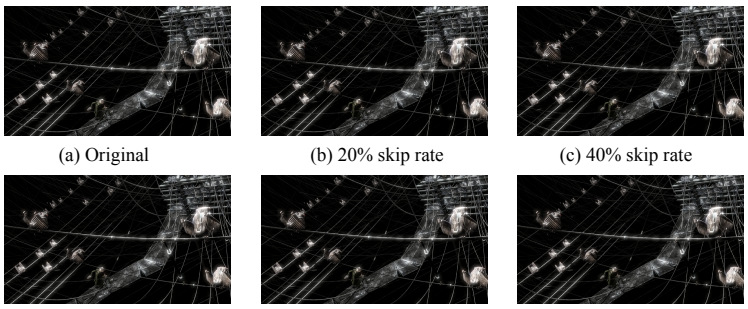
I. Introduction

Since the end of Dennard's scaling [1], there has been limited improvement in transistor speed and energy efficiency. This slows down the steady growth of performance and energy efficiency of general purpose computers. Thus, performance and energy efficiency have become a pressing challenge for modern computing systems. Approximate computing [2]–[4] is gaining a lot of traction as a promising approach to tackle the challenge. Approximate computing trades off accuracy for other benefits such as performance. Many application domains have an inherent tolerance towards inaccuracy. As an example, video encoders are designed to give up perfect accuracy for faster encoding and smaller videos [5]. Machine learning algorithms are designed to produce probabilistic models that are not 100% accurate. Almost all scientific computations (e.g., n-body simulations [6]) are inherently inaccurate in that they are designed to produce an approximation to an ideal result.

Researchers have explored different avenues where approximation techniques can be applied. Loop perforation [2], for example, skips some iterations of a loop to improve performance. Rinard [7] proposes to approximate barrier synchronizations by releasing the processors early. Esmaeilzadeh et al. [4] propose to approximate certain functions of a program using a neural network based accelerator. EnerJ [3] proposes some type qualifiers to declare approximate data with the goal of utilizing the underlying low power storage devices and operations. Samadi et al. [8], [9] propose approximation techniques for data parallel and GPU applications. This paper investigates lock synchronization operations (and the associated critical sections) as a potential source of approximation. Locks are the most widely used synchronization operations for parallel programs. A lock ensures mutual exclusion for shared data. When a thread acquires a lock and other threads attempt to acquire it simultaneously, they need to wait until the first thread releases the lock. This is commonly referred to as *contention* of the lock. Lock contention is a major source of performance problems for parallel programs [10], [11]. This paper explores the potential of approximation to reduce lock contention and thereby, improve performance.

The intuition behind *Approximate Lock* (ALock for short) is that many lock protected critical sections can be occasionally skipped without causing a failure (e.g., crash, deadlock etc.). This is due to the fact that the threads, often, process the same (or similar) data inside a critical section. So, even if a thread occasionally skips some processing of the data, other threads continue to process the same (or similar) data. As a result, the overall inaccuracy caused by the skipped processing remains within a tolerable range. For example, application *X264* of PARSEC [12] uses a lock at line 888 in *frame.c* file. The lock forces a thread to wait until a certain number of pixels of a frame are ready. If the thread occasionally skips the associated critical section, there will be less number of pixels to process. Thus, depending on how many pixels are skipped, the output image might have little or no visible difference. Figure 1 shows the output images for different rate of skipping. The figure shows no visible difference among the images. We can have up to 4.9% performance improvement due to the skipping of this critical section alone. Keep in mind that we use lock skipping and critical section skipping interchangeably throughout the text.

ALock works by making a decision about whether to skip a lock or not. The decision can be based on contention (e.g., whether the lock has a lot of waiting threads or whether a thread has been waiting for a long time for the lock etc.) or some target skip rate. In order to decide where to use ALock, we thoroughly test each of the existing locks using a coverage driven testing tool [13]. The locks that



(a) Original (b) 20% skip rate (c) 40% skip rate

(d) 60% skip rate (e) 80% skip rate (f) Complete skipping

Figure 1. Images produced due to different rate of skipping.

never cause any failure when skipped, are selected as the potential candidates. We replace each of the candidate locks with ALock (one at a time) and determine accuracy loss as well as performance improvement. We also replace multiple candidate locks together and determine the corresponding accuracy loss and performance improvement. Based on the results, we choose the configuration that provides the best performance within a certain accuracy limit. ALock is different from prior work [14], [15] that proposes to eliminate locks completely to trade-off accuracy for performance. *First*, unlike that work, ALock does not eliminate locks alone. ALock skips locks as well as the associated critical sections. As a result, it does not introduce any new data races at all. The issue of not introducing any new data races is particularly important since data races can violate the intuitive sequentially consistent execution model [16], thereby rendering any multithreaded execution impossible to reason about [17], [18]. *Second*, prior work [14], [15] eliminates a lock completely; hence, the approach is applicable only for a few locks and in some cases, requires the code to be rewritten from scratch. ALock, on the other hand, skips a lock and its critical section only under certain conditions. Thus, ALock has a much broader applicability. It does not require the program to be rewritten from scratch. *Last but not least*, unlike the prior work, ALock is dynamic because it decides at runtime when to skip.

This paper presents the *first* proposal to approximate locks dynamically without introducing any new data races. The paper provides a detailed design and implementation of ALock in the standard pthread library. We experimented with 6 applications from PARSEC [12], 3 applications from SPLASH2 [19] and 5 applications from lock-based STAMP benchmark suite [20]. We selected 37 locks as the candidate locks. We determined the best configuration for each application. For an inaccuracy limit of 20%, ALock provides performance improvement for 10 applications, ranging from 1.8% to 164.4%.

This paper is organized as follows: Section II explains the programming model of ALock; Section III explains the design of different ALocks; Section IV points out caveats of ALock; Section V provides the results; Section VI discusses related work, and finally, Section VII concludes.

II. Programming Model

There are two major issues to program with ALock. First, how do we select the appropriate locks/critical sections to approximate and second, how do we use ALock. We are assuming that the program has been already developed using standard locks, we have access to the source code, and we want to modify the code to use ALock. The modification should not a require a complete overhaul of the existing code.

A. Selecting Locks and Critical Sections

Ideally, we would like to focus on locks that are protecting computations of various quantities so that any skipping of those computations does not lead to a failure (e.g., crash, deadlock etc.). However, locks can protect a myriad of shared resources such as pointers, data structures, counters, system resources etc. Moreover, the same lock can be used in multiple places creating multiple different critical sections. Therefore, we need to select locks carefully. There are 2 ways to tackle this issue. Either we can test each lock thoroughly and choose the ones that will not cause any failure when skipped or we can be less rigorous in choosing locks but have some runtime recovery mechanism (e.g., checkpoint and rollback [21]) to avoid any failure when the locks are skipped. In this paper, we take the former approach because it does not require any expensive runtime support for logging and recovery. We leave the later as a future work.

Figure 2 shows the algorithm to select candidate locks for approximation. We start with one lock (say, L) at a time. We skip every instance of L (i.e., every code section where L is used) and its associated critical section with different rates (e.g., 20%, 40%, 60%, 80%, 90%, and 100%). The code in Listing 1 shows how to skip L randomly at a rate of r.

```
1 if (uniform_rand() > r) {
2 // Condition added to skip lock L and its critical
      section at a rate of r
3   pthread_mutex_lock(&L);
4   ...
5   pthread_mutex_unlock(&L);
6 }
```

Listing 1. Modified code to skip a lock and its critical section

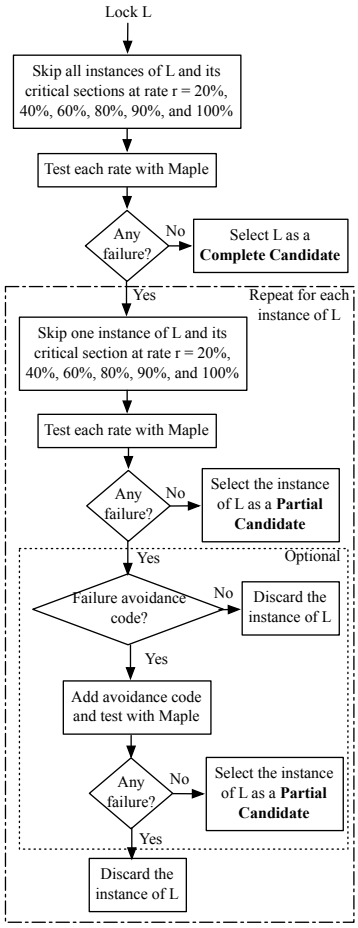


Figure 2. Algorithm to select a candidate lock.

For each instance of r, the resulting program is tested using a tool, called Maple [13]. Maple is a state-of-the-art testing tool with a high coverage and can expose more bugs than other existing tools [22]. If none of the tests causes any failure, we choose L as a *Complete Candidate* for approximation. L is called a complete candidate because we can approximate every instance of L. If any test with Maple leads to a failure, we focus on each instance of L in the code separately. We repeat the following steps for each instance of L. We skip one instance of L and its critical section with different rates. As before, the resulting program is tested with Maple. If no failure occurs during testing, the particular instance of L is chosen as a candidate for approximation. We call L a *Partial Candidate* because only some instance of L can be approximated. However, if any testing with Maple leads to a failure, we try to add some *Failure Avoidance Code* with the critical section. The step with failure avoidance code is optional and is meant to avoid failures due to some common cases that can occur when the critical section is skipped. Section II-A1 describes in detail about failure avoidance code. In any case, as before, we modify the particular instance of L and its critical section to skip at different rates. Failure avoidance code (if exists) is executed when the critical section is skipped. The resulting program is tested with Maple. If any failure occurs, the particular instance of L is discarded from consideration. Otherwise, the instance of L is selected as a partial candidate.

Note that the choice of a testing tool does not affect the design and implementation of ALock. It affects the set of locks that can be replaced with ALock. For example, a model checking [23] based testing tool may find new cases where skipping of a particular lock leads to a failure and thus, discards the lock from consideration. Since Maple has a reasonably high coverage [13], such cases will be very rare. This implies that a lock selected by Maple may lead to a failure only during rare occasions; hence, we argue that such a lock should be considered for approximation too.

1) Failure Avoidance Code: Failure avoidance code is optional. Its purpose is to avoid some common causes of failures when a critical section is skipped. Note that this code is not supposed to rectify any incorrectness that may have occurred by the skipping. It is the best effort code to avoid some failures. An example is shown in Listing 2. The code is taken *Raytrace* of SPLASH2. Inside the critical section, a thread returns $EMPTY$ if it does not find an entry; otherwise, it returns a non empty status code. If there is no failure avoidance code and the critical section is skipped, the later part of the program (not shown here) returns a non empty status code. As a result, the thread tries to process an $EMPTY$ entry and crashes. To avoid this, the avoidance code returns $EMPTY$ when the critical section is skipped; hence, the thread no longer tries to access an $EMPTY$ entry.

```
1  if (uniform_rand() > x) {
2          ... *wpentry = NULL
3    pthread_mutex_lock(...);
4    wpentry = workpool[pid][0];
5    if (!wpentry) {
6      wpstat[pid][0]= EMPTY;
7      pthread_mutex_unlock(...);
8      return EMPTY;
9    }
10   ...
11   pthread_mutex_unlock(...);
12 } else {
13 // Failure avoidance code
14   wpstat[pid][0] = EMPTY;
15   return EMPTY;
16 }
```

Listing 2. When the critical section is skipped, failure avoidance code prevents a failure.

We outline some informal guidelines to write failure avoidance code. *First,* when we add this code, we should not introduce any new data race. This implies that the avoidance code should access local or thread private variables only. For example, in Listing 2, the code modifies `wpstat[pid][0]` which is private to the current thread (i.e., with id `pid`). Hence, the access does not introduce any data race. If the code cannot be added without data races

or additional synchronizations, we discard the particular instance of lock and its critical section from consideration. *Second,* we often need to use stale, constant or initial value for different variables inside the avoidance code. Listing 3 shows how we can use a constant value in the avoidance code. The example is taken from *Canneal* of PARSEC. Here, `seed` is the only shared variable inside the critical section. It is used to initialize a random number generator. `seed` is initialized to 0 and goes up to some maximum value. When the critical section is skipped, the failure avoidance code can use any constant value within that range to initialize the random number generator. This prevents failure because `_rng` is no longer uninitialized or NULL. *Last but not least,* we do not use any avoidance code that requires a major modification (e.g., changing an entire function, class etc.) of the original code. In our experiments, we used failure avoidance code only to avoid null objects and improper status code. We used avoidance code for 2 out of 37 candidate locks.

```
if (uniform_rand() > x) {
  pthread_mutex_lock(...);
  _rng = new MTRand(seed++);
  pthread_mutex_unlock(...);
} else {
// Failure avoidance code
// We can use a constant 1 for seed here
  _rng = new MTRand(1);
}
```

Listing 3. Use of a constant value in failure avoidance code.

B. How to Use an Approximate Lock

Each of the candidate locks and its critical section is transformed to use ALock. An example code is shown in Listing 4. ALock returns $SKIPPED$ when a certain condition is satisfied (details in Section III). In that case, the critical section is completely skipped and failure avoidance code (if exists) is executed. If, on the other hand, ALock does not return $SKIPPED$, the lock is assumed to be acquired, the critical section is executed, and finally, the lock is released.

```
if (ALock_acquire(...) != SKIPPED) {
// Critical section is not skipped
  ...
  ALock_release(...);
} else {
// Critical section is skipped
// Optional failure avoidance code
}
```

Listing 4. How ALock is used?

C. Impact on Semantic

When we use ALock, either the critical section is executed as in the original program or it is completely skipped and failure avoidance code (if exists) is executed. Moreover, the failure avoidance code is intentionally written as data race free. Thus, ALock does not introduce any new data race. In other words, the semantic of multithreaded execution model (i.e., sequential consistency [16]) remains unchanged. However, other bugs such as null pointer dereference, deadlock, use of uninitialized data, memory leak etc. can occur. We use the candidate selection algorithm (Figure 2) to filter out those cases.

III. Design of Approximate lock

We propose 3 types of approximate locks — Counting Approximate Lock (CALock), Timed Approximate Lock (TALock), and Rate-Based Approximate Lock (RALock). We propose the first two to handle high contention. The last one is contention oblivious. The design of these locks is driven by two principles – (i) we want a thread to do additional computations (required for making a decision to skip) when it would, otherwise, wait for the lock (i.e., additional computations become part of the waiting period) and (ii) we want the implementation to be data race fee.

A. Counting Approximate Lock

The intuition behind $CALock$ is that if there are a lot of threads waiting for a lock, we do not want to increase the contention anymore; hence, a new thread will not wait to acquire the lock. In order to decide whether to allow the thread to wait for the lock, we need a threshold for the number of currently waiting threads. If the number of currently waiting threads reaches the threshold, the new thread does not wait for the lock and just skips the corresponding critical section. The threshold can be set to the average number of waiting threads over a period of time. We can also use a fraction, f of the average as a threshold. When the average itself is used as a threshold, f is essentially 1.0. Algorithm 1 shows how $CALock\_acquire$ works.

We start out with an explanation of the variables used in Algorithm 1. tid is the unique id to identify the current thread. $tryCount$ is used to keep track the number of times a thread finds the lock to be contended (i.e., unavailable) and therefore, tries to decide whether to wait or skip. $waitingCount$ is used to keep a total of waiting threads found during the tries. $currentlyWaiting$ keeps track of the total number of threads currently waiting for the lock. $avgWaiting$ denotes the average number of waiting threads during an acquire call. $tryCount$ and $waitingCount$ are accessed per thread basis whereas $currentlyWaiting$ and $avgWaiting$ are shared among the threads. The variables are initialized during lock initialization.

If the lock L is already available, the thread acquires it and returns $ACQUIRED$ status. Otherwise, it checks if the number of currently waiting threads reaches the threshold (i.e., $f \times avgWaiting$). If not, the thread decides to wait for L. The thread updates $currentlyWaiting$ and (its own) $waitingCount$. The thread also periodically (i.e., after every $INTERVAL$ tries) recalculates $avgWaiting$. Note that instead of having a thread deciding to recalculate the average

Algorithm 1 Pseudocode for CALock Acquire

```
1: function CALock_acquire(L)
2:     Let tid be the thread id.
3:     if L is available then
4:         Acquire L
5:         return ACQUIRED
6:     else
7:         skip = TRUE
8:         Increment tryCount[tid]
9:         if currentlyWaiting ≤ f × avgWaiting then
10:            Increment currentlyWaiting
11:            waitingCount[tid] = waitingCount[tid] + currentlyWaiting
12:            skip = FALSE
13:        end if
14:        if callCount[tid]%INTERVAL is 0 then
15:            totalTry = SUM of tryCount of each thread
16:            totalWaiting = SUM of waitingCount of each thread
17:            avgWaiting =
18:                totalWaiting/totalTry
19:        end if
20:        if skip is TRUE then
21:            return SKIPPED
22:        else
23:            Wait and acquire L
24:            Decrement currentlyWaiting
25:            return ACQUIRED
26:        end if
27:    end if
28: end function
```

Algorithm 2 Pseudocode for TALock Acquire

```
1: function TALock_acquire(L)
2:     Let tid be the thread id.
3:     if L is available then
4:         Acquire L
5:         return ACQUIRED
6:     else
7:         Increment tryCount[tid]
8:         if callCount[tid]%INTERVAL is 0 then
9:             totalTry = SUM of tryCount of each thread
10:            totalWait = SUM of waitTime of each thread
11:            avgWaitTime =
12:                totalWait/totalTry
13:        end if
14:        startTime = Read TSC
15:        timeOut = 1
16:        repeat
17:            currentTime = Read TSC
18:            elapsedTime =
19:                currentTime − startTime
20:            if elapsedTime > timeOut then
21:                if L is available then
22:                    Acquire L
23:                    waitTime[tid] = waitTime[tid] + elapsedTime
24:                    return ACQUIRED
25:                else
26:                    timeOut = timeOut × 2
27:                end if
28:            end if
29:        until elapsedTime > f × avgWaitTime
30:        waitTime[tid] = elapsedTime
31:        return SKIPPED
32:    end if
33: end function
```

based on its own $tryCount$, we could have used a shared counter to accumulate the total number of $tryCount$ of all threads and decided based on the counter. We choose not to do so in order to keep the number of shared counters and the associated cache contention at a minimum level. Finally, based on the earlier decision (to skip/wait), the thread either skips the lock or waits to acquire it. Here, waiting is done through a system call (e.g., $futex$ in Linux). In each case, appropriate status code is returned.

$tryCount$, $waitingCount$, $currentlyWaiting$, and $avgWaiting$ can be accessed by multiple threads simultaneously. In order to make the accesses data race free, those variables are read/written using atomic instructions (e.g., fetch-and-increment, exchange etc.). Note that the use of atomic instructions cannot make the process of accumulating both $tryCount$ and $waitingCount$ atomic. Therefore, the calculated average is an approximation to the actual average. This is perfectly reasonable since we use the average as an approximation to the lock's average contention level. Finally, to reduce false sharing due to $tryCount$ and $waitingCount$, we use appropriate padding with them. $CALock\_release$ function is the same as the normal lock release function (e.g., $pthread\_mutex\_unlock$) and hence, is not discussed here.

B. Timed Approximate Lock

The intuition behind $TALock$ is that if a thread waits for a long time, it will not wait any longer i.e., it will skip the corresponding critical section. Like $CALock$, we need a threshold for waiting time after which the thread decides to skip. We use a fraction, f of the average waiting time of the threads as the threshold. Algorithm 2 shows the implementation of $TALock\_acquire$.

$waitTime$ keeps track of waiting time of each thread. Like $CALock\_acquire$, $tryCount$ keeps track of how many times a thread finds the lock to be contended. Both of these variables are accessed per thread basis. When $TALock\_acquire$ is invoked, the thread checks if L is available. If so, the thread immediately acquires it and returns $ACQUIRED$ status. If L is not available, the thread increments its own $tryCount$ (i.e., $tryCount[tid]$). The thread, then, checks if it needs to recalculate the $avgWaitTime$; if so, the the average is calculated in the same way as in $CALock\_acquire$. $TALock\_acquire$ uses Timestamp Counter (TSC). TSC is a 64 bit per processor register to keep track of clock cycles. It is supported by all recent x86 processors [24]. The thread reads the the initial value of TSC. It keeps reading TSC until $elapsedTime$ is greater than the threshold (i.e., $f \times avgWaitTime$). Instead of checking the lock after $f \times avgWaitTime$ has elapsed, the thread uses exponential backoff algorithm to check whether the lock is available in the meantime. If so, it acquires the lock, updates its own $waitTime$, and returns $ACQUIRED$ status. Otherwise, the thread waits for $f \times avgWaitTime$ and then, decides to skip. The thread updates its own $waitTime$ with $elapsedTime$ and returns $SKIPPED$ status.

As in $CALock\_acquire$, $tryCount$, $waitTime$, and $avgWaitTime$ are accessed using atomic instructions to

prevent data races. Moreover, $waitTime$ array is properly padded to avoid false sharing. Finally, $TALock\_release$ remains the same as the normal lock release function.

C. Rate-Based Approximate Lock

Unlike $CALock$ and $TALock$, a thread decides to acquire or skip a lock in $RALock$ randomly based on a target rate. Thus, $RALock$ is contention oblivious. So, it is applicable to both high and low contention scenario. A careful reader might wonder if $RALock$ does not consider lock contention, why would it be useful. $RALock$ can be useful because by skipping critical sections, it can still reduce the amount of work that needs to be done by different threads. Thus, it can contribute to performance improvement.

Algorithm 3 Pseudocode for RALock Acquire

```
1: function GET_RDATA(r)
2:     Allocate rData
3:     rData.callCount = 0
4:     Let rData.bitmap caches probabilistic decision
5:     Let SIZE denotes the length of rData.bitmap
6:     for i = 0 to SIZE − 1 do
7:         if uniform_rand() < r then
8:             rData.bitmap[i] = 0
9:         else
10:            rData.bitmap[i] = 1
11:        end if
12:    end for
13:    return rData
14: end function

15: function RALOCK_ACQUIRE(L, rData)
16:    Increment rData.callCount
17:    if rData.bitmap[rData.callCount%SIZE] is 1 then
18:        Acquire the L similar to a normal acquire function
19:        return ACQUIRED
20:    else
21:        return SKIPPED
22:    end if
23: end function
```

Algorithm 3 shows the pseudocode of $RALock$. Unlike $CALock$ and $TALock$, a thread calling $RALock\_acquire$ decides whether to skip the lock, even before checking if the lock is free. Therefore, we need to make the decision very quickly. A naive approach would use some uniform random number generator inside $RALock\_acquire$ to skip the lock randomly at a certain rate. However, calling the random number generator adds a significant overhead, especially when the lock is free and the thread decides to acquire it. Therefore, we cache the random probabilistic decisions (for some number of calls) in $bitmap$ and use those decisions repeatedly. $rData$ is the meta data associated with $RALock$. $rData$ has two elements - $bitmap$ for caching the decisions and $callCount$ to keep track of how many times $RALock\_acquire$ has been called. A programmer uses get_RData to allocate and initialize $rData$. The function initializes $callCount$ to 0 and $bitmap$ with 1 or 0 with probability r. When $RALock\_acquire$ is called, the thread increments $callCount$ associated with $rData$. It, then, checks if the proper bit in $bitmap$ is set. If so, the thread acquires lock L, just like a normal lock acquire function. On the other hand, if the bit is clear, the thread skips the lock. In any case, appropriate status is returned.

$callCount$ and $bitmap$ are both accessed by multiple threads. However, $bitmap$ is read-only after initialization. Hence, it does require atomic instructions. $callCount$, on the other hand, is accessed using atomic instructions to avoid data races.

D. Selecting an Optimal ALock

Each candidate lock can be approximated using one of the three ALocks. To find an optimal one, we modify the program to use a particular ALock in place of the original lock and enumerate over different design parameters. For example, we use $CALock$ with different values of f i.e., $f = 0.1, 0.2, 0.3, ..., 1.0$ and collect data for performance improvement and accuracy degradation. Similarly, we use $TALock$ with $f = 0.1, 0.2, 0.3, ..., 1.0$. For $RALock$, we, first, use different values of r (e.g., $r = 10\%, 20\%, 30\%, ..., 100\%$) and collect data for performance improvement and accuracy degradation. Then, we use regression analysis (e.g., Quadratic Polynomial Regression) to predict an interval for r that can lead to a (positive) performance improvement (note that a negative performance improvement is equivalent to a slow down of the program). As an example, let us assume that the regression analysis predicts that any $r \in [70\%, 90\%]$ leads to a performance improvement. We run experiments with all values of r in progression of 1% (e.g., 70%, 71%, 72%, ..., 89%, 90%) within the predicted interval and collect data for performance improvement and accuracy degradation. Whichever ALock provides the highest performance improvement without degrading accuracy below a predefined threshold is selected as the optimal ALock.

We would like to elaborate a few issues regarding the above mentioned selection algorithm. *First,* we use regression analysis only for $RALock$. We do not use it for the other two ALocks, because those ALocks tend to provide monotonically increasing performance improvement and accuracy degradation. Therefore, experimenting with a fixed set of values is found to be enough. *Second,* for $RALock$, we predict an interval for performance improvement, not for accuracy degradation, because performance is found to be more predictable than accuracy. *Finally,* in order to get a statistically significant result, we experiment with each design choice multiple times (e.g., 10 times) and take an average for performance improvement and accuracy degradation.

E. Composability

With ALock, when a thread acquires a lock, the thread acquires it using the same algorithm used in a normal lock acquire function (e.g., $pthread\_mutex\_lock$ function). Moreover, the release function remains the same. Therefore, ALocks can easily co-exist with other ALocks or normal locks.

IV. Caveats

First, as mentioned in Section II-A, the set of locks chosen for approximation depends on the choice of a testing tool. We used Maple [13] because it is open source and a state-of-the-art coverage driven testing tool. A more comprehensive (e.g., model checking based) testing tool may discard some of the candidate locks that Maple selected. However, the design, implementation, and programming model of ALock is orthogonal to the choice of a testing tool. *Second,* failure avoidance code is the best effort code to avoid some of the failures that occur due to skipping of a critical section. It is an optional step that a programmer can choose to ignore. *Third,* we have not considered adhoc synchronizations found in many large applications.

V. Evaluation

A. Experimental Setup

We ran experiments on a 2 socket 12 core Intel Xeon 2.00 GHz system with 32GB memory. We implemented our locks in glibc pthread library. We used gcc 4.4.7 with -O3 optimization. For 16 threaded executions, we ran experiments on a 2 socket 16 core Intel Xeon 2.00 GHz system. We used 6 applications from PARSEC [12], 3 applications from SPLASH2 [19] and 5 applications from STAMP [20] benchmark suite. We manually converted STAMP benchmarks to use locks. By default we used native input set with 8 threads. For 16 threads (Section V-D3), we used simsmall, simlarge and native input sets. Table I shows the accuracy metric used for different applications. We considered 20% accuracy loss as acceptable.

App.	Accuracy Metric
Canneal	Relative distance of routing cost.
X264	Video encoding quality.
Bodytrack	Relative distance between poses vector.
Ferret	Difference between the number of similar images.
Fluidanimate	Relative distance between particle position, acceleration, and force.
Dedup	Compression ratio and decoding ability.
Raytrace	Relative distance between pixel value.
Radiosity	Relative distance between pixel value.
Fmm	Relative distance between particle position.
kmeans	Number of cluster changed or dropped
genome	Comparison of produced sequence with given gene
ssca2	Difference between number of undirected edges
Intruder	Difference between number of network attacks
vacation	Difference between number of transactions

Table I
Accuracy metric.

B. Characterization of Locks and Critical Sections

Table II shows the candidate locks. These locks are selected by our algorithm in Section II-A. The 3rd and 4th columns show how many profile and test runs were performed by Maple [13]. At the high level, the tool collects some "dangerous" interleaving patterns during profile runs and changes thread scheduling during test runs to expose them. The tool performed anywhere from 1 to 6137 profile runs and 1 to 2915 test runs. The 5th column shows the type of candidate locks. There are 23 complete candidate locks. Others are partial candidates. The 7th and 8th columns show the average number of waiting threads and the average waiting period for the threads. These two values indicate the contention level of the original lock. Most of the locks do not have a significant lock contention. Raytrace, Genome, Ssca2, and Intruder have some contention for candidate locks. The last column shows the rate interval predicted for $RALock$. '-' indicates that the regression analysis did not find any rate interval where there can be some performance improvement. The analysis predicted intervals for 15 locks.

C. Overhead of ALock

In order to determine the overhead of ALock, we designed a small kernel. The kernel creates 4 (for low contention scenario) or 8 (for high contention scenario) threads. Each thread calls lock acquire and release functions 1 million times in a loop. We used $pthread\_mutex\_lock$ and $pthread\_mutex\_unlock$ in our baseline execution. For $CALock$ and $TALock$, we experimented with different $INTERVAL$ (e.g., 100, 500, 1000, 5000, and 10000) and used the average waiting threads/periods as threshold. For $RALock$, we experimented with different skip rates (e.g., 20%, 40%, 60%, 80%, and 100%). In order to isolate the impact of skipping, we used two versions of each ALock - one where the acquire function applies the algorithm of Section III but always returns $ACQUIRED$ status (*no skip*) and one where the acquire function returns either $ACQUIRED$ or $SKIPPED$ status according the algorithm (*with skip*). Figure 4 shows the data.

Figure 4(a) shows overhead for $CALock$. When there is no skipping during high contention, $CALock$ has less than 1% overhead for every interval. During low contention, the overhead stays below 1% except at 10000 interval. At that interval it rises to 1.6%. When skipping of critical sections is enabled, $CALock$ performs better at high contention. Overall, having a high interval is not effective. So, we chose 100 as the default value of $INTERVAL$. For that interval, $CALock$ has a maximum overhead of slightly more than 1%. Figure 4(b) shows overhead for $TALock$. When there is no skipping, $TALock$ performs more or less the same during every interval and contention level. When skipping of critical sections is enabled, $TALock$ actually provides some performance improvement (negative overhead means speed up). The improvement is more for high contention. This also implies that $TALock$ is able to skip more critical sections compared to $CALock$. We chose the same default value of $INTERVAL$ for $TALock$. Figure 4(c) shows overhead for $RALock$. During every contention level, $RALock$ has around 1% overhead when there is no skipping. When skipping is enabled, $RALock$ provides linear performance improvement. Overall, $RALock$ provides more improvement than both $TALock$ and $CALock$.

App.	Lock (Id)	Maple Related		Candidate Type	Description	Avg waiting threads	Avg waiting period (μsec)	Predicted rate interval
		# Profiles	# Tests					
Canneal	seed_lock (1)	8	48	Part.*	Protects seed for random number generation. Corrective code uses a fixed seed.	0.5	1125	60%-80%
X264	frame (2)	5	164	Comp.	Forces a thread to wait until a certain number of pixels are ready.	0	0	90%-100%
Bodytrack	mDatalock (3)	6	46	Part.	Forces a thread to wait for a new image when the buffer is empty.	0	0	-
	l (4)	9	26	Part.	Protects a counter.	0	0.013	90%-100%
Ferret	mutex (5)	5	168	Part.	Protects queuing operation.	0	0	1%-20%
Fluidanimate	ipar (6)	5	1	Comp.	Protects calculation of force and acceleration.	0	0	-
	iparNeigh (7)	7	7	Comp.	Protects calculation of force and acceleration.	0	0	-
Dedup	chunk_header_lock (8)	10	586	Part.	Protects file write operation.	0	0	-
	Ht_lock (9)	8	498	Part.*	Protects free operation of reference counted object.	0.001	0.002	25%-38%
Raytrace	ridlock (10)	9	698	Comp.	Protects a single variable increment operation.	0.002	0.01	-
	memlock (11)	6	225	Part.	Protects free operation.	0.002	0.22	10%-100%
Radiosity	elem_ev1_ev_lock (12)	27	1	Comp.	Protects rgb calculation.	0	0	-
	elem_ev2_ev_lock (13)	15	1	Comp.	Protects rgb calculation.	0	0	-
	elem_ev3_ev_lock (14)	23	1	Comp.	Protects rgb calculation.	0	0	-
	ev_ev_lock (15)	16	1	Comp.	Protects rgb calculation.	0	0	-
	elem_elem_lock (16)	7	1	Part.	Protects list insert operation.	0	0.002	1%-5%
	global_bsp_tree_lock (17)	25	1	Comp.	Protects bsp tree initialization and traversal.	0	0	1%-10%
	global_avg_radiosity_lock (18)	36	1	Part.	Protects rgb calculation.	0	0	-
	global_free_interaction_lock (19)	1	1	Part.	Protects free operation of list elements.	0.008	0.0136	-
	e_elem_lock (20)	6137	1	Part.	Protects rgb calculation.	0	0	1%-10%
Fmm	io_lock (21)	2	1	Comp.	Protects I/O operation	0	0	-
	cb_exp_lock_index (22)	4	1	Comp.	Protects data used in calculating multipole expansion.	0	0	-
	b_exp_lock_index (23)	2	1	Part.	Protects data used in calculating multipole expansion.	0	0	-
	pb_exp_lock_index (24)	2	1	Part.	Protects data used in calculating multipole expansion.	0	0	-
	dest_exp_lock_index (25)	2	1	Comp.	Protects data used in calculating multipole expansion.	0	0	-
Kmeans	lock_1 (26)	9	171	Comp.	Updates objects of new cluster centers	0	0	-
	lock_2 (27)	10	67	Comp.	Update task queue after checking for convergence	0	0	-
	lock_3 (28)	3	1	Comp.	Update a global variable which is checked against threshold that Determines whether to continue the clustering	0	0	-
Genome	lock_1 (29)	9	2828	Comp.	Insert segments into hashtable	0.00028	46.589	1%-100%
	lock_4 (30)	9	1653	Comp.	Insert hashes of segment substrings into hashtable	0.454	0.0162	-
	lock_5 (31)	11	2915	Comp.	Insert constructEntries and endInfoEntries into hashtable	0.0025	1.0736	-
	lock_6 (32)	14	1649	Comp.	Match ends to starts by using hash-based string comparison	0.1434	1.0291	-
Ssca2	lock_2 (33)	8	448	Comp.	Use auxiliary array to store the undirected edge	0.1560	0.8496	15%-30%
Intruder	lock_2 (34)	5	290	Part.	Process network packet for finding number of attacks	0.1350	17.411	1%-10%
Vacation	lock_1 (35)	10	203	Comp.	Process add customer and car reserve transaction	0.0094	140.71	15%-30%
	lock_2 (36)	9	304	Comp.	Process flight reserve transaction	0	0	15%-30%
	lock_4 (37)	6	201	Comp.	Process reserve room transaction	0	0	15%-30%

Table II
DESCRIPTION OF THE LOCKS THAT CAN BE SKIPPED. COMP. = COMPLETE, PART. = PARTIAL, AND PART.* = PARTIAL WITH FAILURE AVOIDANCE CODE.

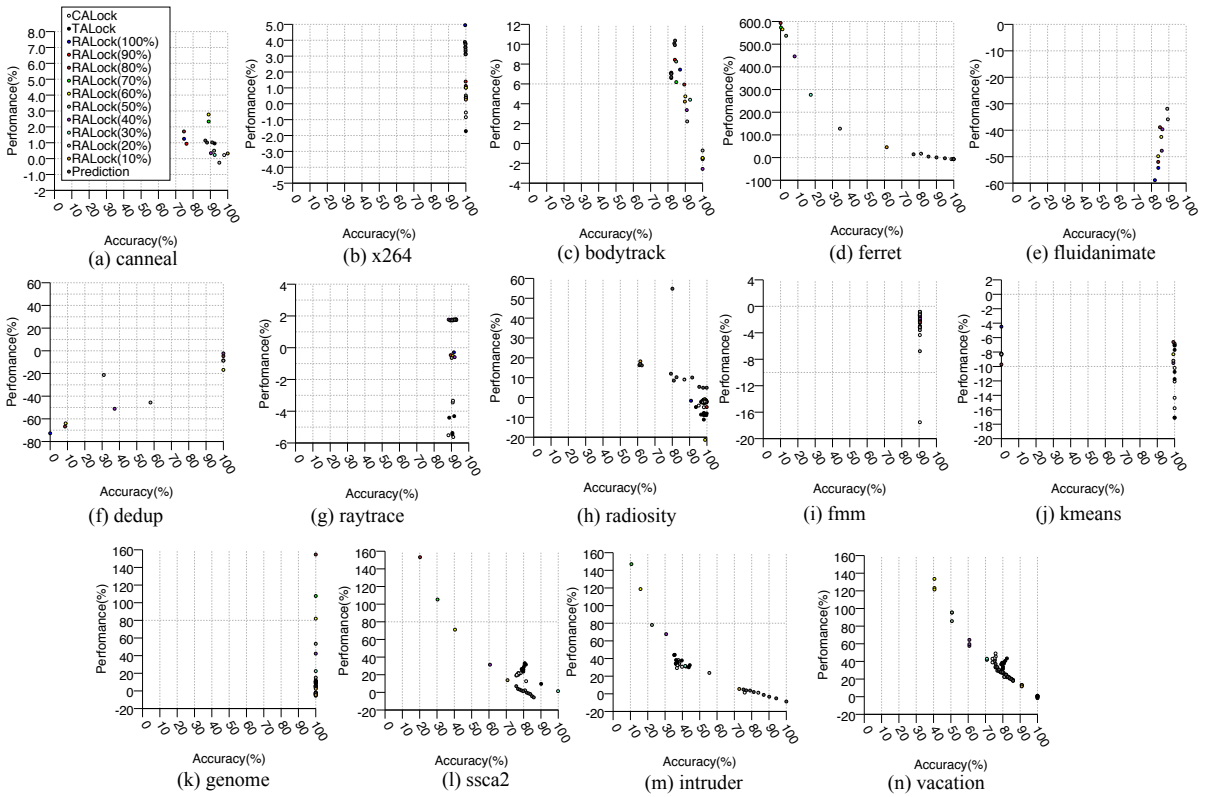
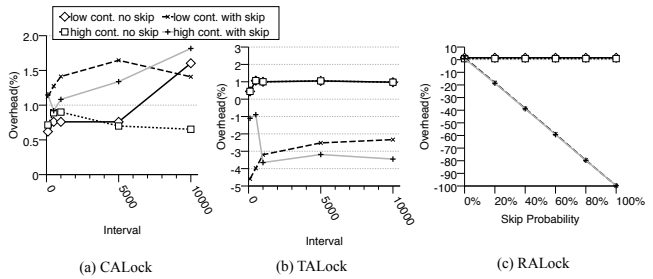


Figure 3. Single lock approximation results



(a) CALock (b) TALock (c) RALock

Figure 4. Overhead of different ALocks.

D. Accuracy & Performance

1) Single Lock Approximation: For each candidate lock, we experimented with all three types of ALocks. For $CALock$ and $TALock$, we used $f = 0.1, 0.2, ..., 1.0$ i.e., 10 different f values. For $RALock$, we used $r = 10\%, 20\%, ..., 100\%$ as well as the predicted interval values. We used the original program with 8 threads and native input as the baseline. Figure 3 shows the accuracy vs performance plot for Canneal (3(a)), X264(3(b)), Bodytrack(3(c)), Ferret(3(d)), Fluidanimate(3(e)), Dedup(3(f)), Raytrace(3(g)), Radiosity(3(h)), Fmm(3(i)), Kmeans(3(j)), Genome(3(k)), Ssca2(3(l)), Intruder(3(m)) and Vacation(3(n)). Any point with accuracy at least 80% is acceptable. Within this accuracy constraint, we observed performance improvement in Canneal, X264, Bodytrack, Ferret, Raytrace, Radiosity, Genome, Ssca2, Intruder, and Vacation.

Table III shows the best lock for each program that has some performance improvement. The best lock for a program is the one which improves performance without sacrificing accuracy by no more than 20%. Overall, we observed performance improvement from 1.8% to 154.6% and accuracy from 80.1% to 100%. We observed more than 10% performance improvement in 5 applications. In most of the cases $RALock$ provides the best solution. $CALock$ and $TALock$ are designed to handle high contention but our candidate locks do not have that much contention. That is why, $RALock$ works the best for these locks. In 2 cases, $TALock$ performed better than others. This happens when there is a moderate to high contention.

App.	Lock	Best $ALock$	Accu.(%)	Perf. Imp.(%)
Canneal	1	$RALock$ (60%)	89.0%	2.8%
X264	2	$RALock$ (100%)	99.6%	4.9%
Bodytrack	4	$RALock$ (92%)	84.1%	10.4%
Ferret	5	$RALock$ (5%)	81.1%	17.3%
Raytrace	11	$RALock$ (90%)	91.5%,	1.8%
Radiosity	17	$RALock$ (3%)	80.1%	54.8%
Genome	29	$RALock$ (80%)	100.0%	154.6%
Ssca2	33	$TALock$ (0.1)	81.3%	31.5%
Intruder	34	$RALock$ (6%)	81.2%	2.1%
Vacation	35	$TALock$ (0.3)	82.4%	43.4%

Table III
BEST PERFORMING LOCKS WITH AT LEAST 80% ACCURACY.

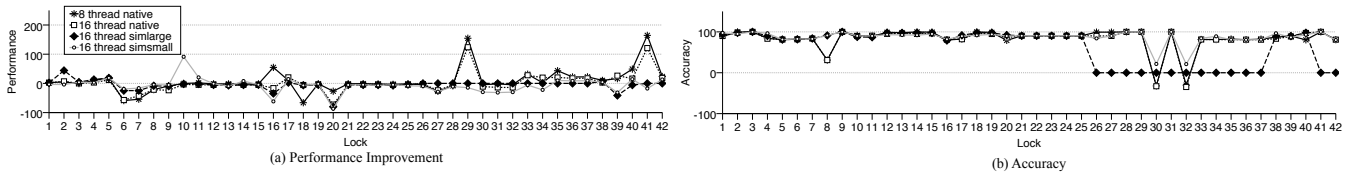
Case Studies: We give a brief overview of some of the locks in Table III. Specifically, we look into the ones with performance improvement more than 10%. Lock 29 of Genome gives the highest performance improvement of 154.6%. Genome implements a gene sequencing program. In the first step, duplicate segments are removed using hash-set which is protected by lock 29. When we skip the lock, some of the segments are dropped from hash-set. Thus, the search needs to be done on less number of segments. But the segments are still sufficient enough to construct the source gene. Thus, we get the same accuracy with improved performance. Lock 17 of Radiosity results in 54.8% performance improvement. The lock protects tree generation process. Skipping the lock drops some nodes from the tree. As a result, the threads need to process less number of nodes. Lock 35 of Vacation results in 43.4% performance improvement. This lock protects customer addition and car reservation transaction. Skipping this lock skips some number of transactions and hence, performance improves. Both lock 17 and 35 take a hit on accuracy too. Lock 33 of Ssca2 yields 31.5% performance improvement. This lock protects edge creation in a graph. Skipping it causes the graph to have less number of edges. As a result performance is improved but accuracy is also degraded. Lock 5 of Ferret provides 17.3% performance improvement. This lock is similar in functionality to lock 17 and hence, the same explanation also applies here. Finally, lock 4 of Bodytrack provides 10.4% performance improvement. Skipping this lock drops work units to be processed by other threads. So, performance is improved.

2) Multiple Lock Approximation: For the applications that have multiple candidate locks, we combine the ALocks that do not violate the accuracy constraint. We can form such combination for Fluidanimate, Bodytrack, Raytrace, Radiosity, Fmm, Kmeans, Genome, and Vacation. The applications that have only 2 candidate locks, can have only 1 combination. In case of more than 2 candidate locks, we experimented with all possible combinations of locks. Since $RALock$ has been found to work the best, we experimented with $RALock$ in majority of the cases. Figure 6 shows the results for Fluidanimate (6(a)), Bodytrack (6(b)), Raytrace (6(c)), Radiosity (6(d)), Fmm (6(e)), Kmeans (6(f)), Genome (6(g)), and Vacation (6(h)).

Table IV shows the best combination for each application where we observed performance improvement without any violation of the accuracy constraint. Multiple lock approximation can improve performance by 10.3% up to 164.4%. We also observed that in all cases, combinations involving $RALock$ provides the best solution.

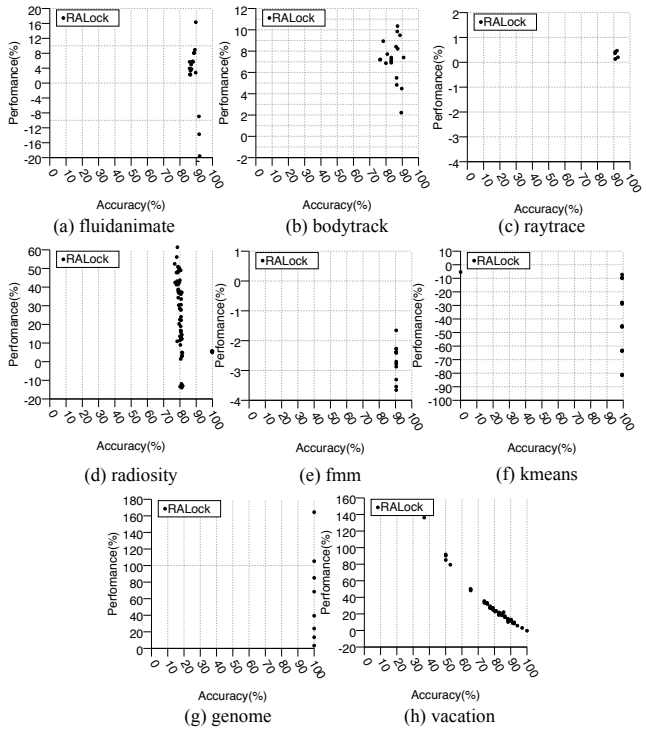
App.	Lock	Best $ALock$	Accu.(%)	Perf. Imp.(%)
Bodytrack	3,4	$RALock$ (91%)	87.1%	10.3%
Fluidanimate	6,7	$RALock$ (100%)	89.7%	16.3%
Radiosity	13,14, 15,19	$RALock$ (7%)	80.5%	49.0%
Genome	29,31	$RALock$ (80%)	100.0%	164.4%
Vacation	35,36	$RALock$ (11%)	80.9%	23.7%

Table IV
BEST PERFORMING COMBINATIONS OF LOCKS WITH AT LEAST 80% ACCURACY.



(a) Performance Improvement (b) Accuracy

Figure 5. (a) Performance and (b) accuracy results across inputs.



(a) fluidanimate (b) bodytrack (c) raytrace

(d) radiosity (e) fmm (f) kmeans

(g) genome (h) vacation

Figure 6. Multiple lock approximation results

3) Results with Different Inputs: To show that ALocks and their combinations provide consistent results, we take the best performing ALock for each candidate lock and their combinations and ran experiments on a 16 core machine with different inputs. Figure 5 shows the results. Locks marked with id from 38 to 42 are the combinations shown in Table IV in that order. Most of the locks were consistent in terms of performance improvement and accuracy degradation. Lock 10, 16, 18, 20, 29, 39, and 41 showed variation in performance whereas locks from 26 to 37 and 8, 41, 42 showed some variation in accuracy. This suggests that while most locks work well with ALock, some require to adapt rate/fraction of ALock dynamically based on a feedback loop. We leave such an advanced design for future.

VI. Related Work

There has been a growing interest in the field of approximate computing. Loop perforation [2] identifies tunable loops of a program and transforms them to execute a subset of iterations. Using greedy search it identifies patterns of loop which are suitable for perforation. Rinard proposed to skip tasks as a way to make applications more robust in the face of faults [25]. Such skipping can translate in accuracy degradation. Parrot Transformation [4] leverages hardware accelerator to produce approximate results. It replaces programmer identifed code segment with a trained neural network that mimics the region of code. Apporoximate computing introduces opportunities to alleviate synchronization bottlenecks in parallel programs. One such approach is early phase termination [7]. It eliminates the idling of processors at barrier synchronization points by terminating the parallel phase when there are too few of them remaining. This technique applies statistical model to characterize the effect of terminating tasks. Misailovic et al. [26] proposed the use of loop perforation for quality of service profiling to help developers find subcomputations that can be replaced with new (and potentially less accurate) subcomputations that deliver significantly increased performance in return for acceptably small quality of service losses. Hoffmann et al. [8], [27], [28] proposed system that dynamically adjust approximation level to optimize performance, power and energy consumption. We believe ALock can complement the existing approximation techniques.

VII. Conclusion

This paper explored the potential for approximating locks. This is the *first paper* to approximate locks dynamically without introducing any data race. We started out with the observation that many applications can tolerate occasional skipping of computations done inside a critical section protected by a lock. To exploit this opportunity, we proposed 3 types of ALock. The thread executing ALock checks if a certain condition (e.g., high contention, long waiting time) is met and if so, the thread returns without acquiring the lock. Using ALock, we converted some selected critical sections so that those sections are skipped when ALock returns without acquiring the lock. We experimented with 14 programs from PARSEC, SPLASH2, and STAMP benchmarks. We found a total of 37 locks that can be transformed into ALock. ALock provides performance improvement for 10 applications, ranging from 1.8% to 164.4%.

Acknowledgements

We thank the anonymous reviewers for their valuable feedback. This work is supported by University of Texas at San Antonio and NSF under Grant No. 1319983.

REFERENCES

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, June 2011.

[2] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *FSE*, September 2011.

[3] 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 *PLDI*, June 2011.

[4] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in *MICRO*, December 2012.

[5] "x264," http://www.videolan.org/x264.html.

[6] S. Aarseth, *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge University Press, 2003.

[7] M. C. Rinard, "Using early phase termination to eliminate load imbalances at barrier synchronization points," in *OOPSLA*, 2007.

[8] 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*, ser. MICRO-46. New York, NY, USA: ACM, 2013, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540711

[9] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 35–50. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541948

[10] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield, "Analyzing lock contention in multithreaded applications," in *PPoPP*, February 2010.

[11] R. Gu, G. Gin, L. Song, L. Zhu, and S. Lu, "What change history tells us about thread synchronization," in *FSE*, August 2015.

[12] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, October 2008.

[13] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, 2012.

[14] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, ser. RACES '12, 2012, pp. 41–50.

[15] M. Rinard, "Parallel synchronization-free approximate data structure construction," in *The 5th USENIX Workshop on Hot Topics in Parallelism*. Berkeley, CA: USENIX, 2013.

[16] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Tran. on Comp.*, July 1979.

[17] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *Western Reseach Laboratory-Compaq. Research Report 95/7*, September 1995.

[18] S. Adve, "Data races are evil with no exceptions: Technical perspective," *Commun. ACM*, vol. 53, no. 11, Nov. 2010.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995, pp. 24–36.

[20] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *IEEE International Symposium on Workload Characterization*, Sept 2008, pp. 35–46.

[21] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec: Efficient online multiprocessor replayvia speculation and external determinism," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV, 2010.

[22] M. Musuvathi, S. Qadeer, and T. Ball, "CHESS: A systematic testing tool for concurrent software," Microsoft Research, Tech. Rep. MSR-TR-2007-149, November 2007.

[23] C. Baier and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[24] Intel, "How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures," https://www-ssl.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-32-ia-64-benchmark-code-execution-paper.pdf.

[25] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 324–334. [Online]. Available: http://doi.acm.org/10.1145/1183401.1183447

[26] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of service profiling," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 25–34. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806808

[27] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950390

[28] H. Hoffmann, "Jouleguard: Energy guarantees for approximate applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 198–214. [Online]. Available: http://doi.acm.org/10.1145/2815400.2815403