2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems

High Performance Approximate Computing by Adaptive Relaxed Synchronization

Bashima Islam[*], Faysal Hossain Shezan[†] and Rifat Shahriyar[‡]
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
bashimaislam@gmail.com[*], *faysalhossain2007@gmail.com*[†], *rifat@cse.buet.ac.bd*[‡]

***Abstract*—Approximate computing has the potential to provide approximate results with user defined error bound faster than conventional computing. Relaxed synchronization is one of the many ways to achieve approximate computation. Researchers in this area primarily focus on programming languages like C/C++, but languages like Java are still largely overlooked. In Java, generally full synchronization can be achieved by using synchronized keyword for method and block level or by using various locks of Java concurrency utilities framework. We provide a detailed performance evaluation of these different mechanisms to achieve full synchronization in Java. We introduce an adaptive locking mechanism using existing locks of Java concurrency utilities framework to provide relaxed synchronization for Java to be used for approximate computing. Our novel relaxed synchronization based framework achieved one of the important outcomes of approximate computing, better performance.**

***Keywords*-Approximate Computing; Java; Concurrency; Synchronization;**

I. Introduction

Computers were first designed to solve problems by calculating precisely. Though for any program accurate result is considered to be an obligatory feature, achieving precise outcome sometimes requires significant computing resources. Runtime or execution time is generally considered the most rational estimate of performance. However, in domains like big data, data analysis, image processing, we can trade-off between accuracy and performance as there are possibilities of not requiring precise computation. Nowadays, many applications in these domains are being designed to use statistical and probabilistic methods for computing approximate output from noisy input. Reduced runtime implies better performance, but is a critical goal to achieve. Accepting approximate result rather than the accurate one is taking one step closer to this goal. For example, in a face recognition system, it is very common to consider 70-80% similarity as the match. In this case, we can afford a small error because 100% accuracy is not a requirement. To illustrate, let us consider that two images of 128 pixels are compared to do a face recognition. If 100 pixels of the test image matches with the reference image we consider it as a match, we do not need to match all 128 pixels. Compromising accuracy here does not affect the outcome but it reduces the runtime as some pixels are not being compared.

Multi-core processors are widely used across many application domains. Performance requirements are the primary reason for shifting to multi-core processors. To enable any application for multi-core systems, there must be a reliable way to find and eliminate race conditions. To avoid race conditions, access to shared resources and memory must be synchronized. Synchronization is a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as the critical section. Several researchers are relaxing synchronization to introduce approximation in multi-core systems. The errors generated by relaxing synchronization imposes a very little effect on overall outcome due to the characteristics of such systems.

The researchers have concentrated mainly on programming languages like C, C++ for approximate computing. Languages like Java are often ignored to support approximate computation. In Java, synchronization is achieved by synchronized keyword as well as various locks provided by Java concurrency utilities framework. We have developed an adaptive model of relaxed synchronization for approximate computing by utilizing these locks where users will define their desired error level and the system will adapt accordingly. We make the following contributions:

- We introduce a framework for high-performance approximate computing in Java by relaxing synchronization. We propose an adaptive locking mechanism that will allow users to define their desired error level.
- We perform a comprehensive analysis of the different full synchronization mechanisms in Java.
- We find two unusual scenarios while using Lock and Thread class in Java.
- We use our framework for developing an approximate version of a well-known data structure and compare its performance.

The remainder of this paper is organized as follows. Section II gives a brief overview of the related works. In section III, we provide our detailed analysis of different synchronization mechanisms in Java. Section IV discusses the design and implementation of our framework. Section V describes our experimental methodologies and section VI provides our experimental results. Finally, section VII concludes proposing some future works and providing a summary of our whole work.

II. Related Works

This section overviews the necessary approximate computing background on which we build.

A. Approximate Computing

Recently a growing number of applications are error-tolerant due to the flexible requirement of approximate correctness. Data structures involved in approximate computation can drop some inserted elements or retain some deleted elements. These do not crash the program rather produce a state consistent enough to deliver acceptable output. There have been extensive studies on software and hardware level approximate computing that trades off computational accuracy for high performance. Rinard [1] introduced approximate tree and array building blocks with associated approximate data structure construction algorithms. Baek and Chilimbi [2] proposed a system named Green, which enables programmers to approximate expensive functions and loops. Zhang et al. [3] proposed an approximate computing framework for iterative method and a quality estimator to provide quality guarantees. The ACCEPT framework by Sampson et al. [4] is an auto-tuning system that automatically chooses the best approximation strategies and uses a feedback mechanism that helps to improve for better approximation opportunities. Hoffmann et al. [5] presented code perforation for trading accuracy in return for performance. Approximate hardware design can achieve both energy efficiency and faster execution than the exact ones. Pekhimenko et al. [6] proposed a recovery-free hardware/software system design for applying approximate computing to address the problem of memory latency due to cache misses.

B. Relaxed Synchronization

Relaxed synchronization is one of the methods to achieve approximate computing in multi-core systems. Rinard [7] presented a set of unsynchronized techniques to achieve approximate parallel computing by eliminating synchronization overhead, parallelism reduction, and failure propagation when a thread fails to execute a synchronization operation. Renganarayana et al. [8] restructured code of the identified parallel region and pragmatically selected profitable degree of relaxation to exploit relaxed synchronization. Gustedt and Jeanvoine [9] tried to control access to shared or distributed resources using ordered read-write lock (ORWL) library that works well on multi-core machines and clusters.

C. Approximate Computing in Java

Most of the works on approximate computing are done in functional languages like Haskell. Among a few works in Java, Paleczny et al. [10] presented a locking protocol, Relaxed-Lock. The Relaxed-Lock uses only one machine word in the object header and requires only one atomic compare and swap to lock a monitor, and no atomic instructions to release a monitor. Newton et al. [11] presented an adaptive data structure more scalable and lock-free than the standard ones while preserving its benefits in Haskell. They also compared their adaptive data structures with non-adaptive ones in both Haskell and Java.

III. Analysis

This section provides our detailed analysis on different synchronization mechanisms in Java with their advantages and disadvantages. This section also provides some interesting findings related to Java synchronization.

A. Threads in Java

Java provides built-in support for multithreaded programming. Multithreading enables to write efficient programs that make maximum use of the processing power available in the system. There are different ways to create a thread in Java. The most common ones are by extending Thread class and by implementing the Runnable interface. Another way of creating threads is using thread pool that is becoming popular with the programmers. Thread pool is useful to limit the number of threads running in the application. Instead of starting a new thread for every task to execute concurrently, the task can be passed to a thread pool. As soon as the pool has any idle threads the task is assigned to one of them and executed. The major benefit of using thread pool is that the programmers do not need to keep track of the threads, schedule the waiting threads or assign tasks to the threads. This makes the job of the programmers easier.

B. Thread Synchronization

Synchronization in Java is an important concept since Java is a multithreaded language where multiple threads run in parallel to complete program execution. To avoid race conditions in multithreaded systems, access to shared resources and memory must be synchronized. Synchronization is a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment. Synchronization in Java will only be needed if shared object is mutable. If the shared object is either read-only or immutable object, then synchronization is not needed, despite running multiple threads. JVM guarantees that Java synchronized code will only be executed by one thread at a time. When a mutable shared object is being updated that portion of the execution is called critical region and the synchronization is ensured in this region to avoid any corruption of state or any kind of unexpected behavior. Starvation, deadlock, priority inversion, busy waiting etc. are some of the major problems caused by the lack of synchronization. The most common ways of achieving thread synchronization in Java are synchronized keyword based synchronization and lock based synchronization. Their performance depends on the type of programs.

Java synchronized keyword provides different functionalities essential for multithreaded concurrent programming. They are: a) synchronized keyword provides locking, which ensures mutually exclusive access to the shared resource and prevents data race, b) synchronized keyword also prevents reordering of code statement by the compiler which can cause a subtle concurrent issue without the use of synchronized keyword, and c) synchronized keyword involves locking and unlocking and before entering into synchronized method or block thread needs to acquire the lock, at this point it reads data from main memory than cache and when it releases the lock, it flushes write operation into main memory which eliminates memory inconsistency errors.

Java concurrency utilities framework provides four types of Lock. They are: a) Lock, b) ReadWriteLock, c) ReentrantLock, and d) ReentrantReadWriteLock. Lock is the simplest of them all which can be acquired and released by using a single variable. In ReadWriteLock, multiple read locks can be held simultaneously unless the exclusive write lock is held. ReentrantLock and ReentrantReadWriteLock are like the previous two with the flexibility to try for a lock without blocking. Lock provides all the features of synchronized keyword with different ways to create different conditions for locking, providing time out for a thread to wait for the lock. Some of the important methods are lock() to acquire the lock, unlock() to release the lock, tryLock() to wait for the lock for a certain period of time.

C. Analysis of Different Multithreaded Workers

We examine the execution of three different multithreaded workers mentioned in Figure 1. Among these workers, *NoSyncWorker* has no synchronization among the threads. *SyncWorker* and *LockWorker* ensure proper synchronization among the threads. We execute them for a variable number of threads for 100000 iterations using both Java 7 and 8. We can see from Figure 2 that *NoSyncWorker* performs significantly faster than both *SyncWorker* and *LockWorker* but its accuracy is very low due to lack of synchronization. That means without any synchronization the accuracy will be very low but the runtime will be very fast. On the other hand, with proper synchronization, the accuracy will be perfect but the runtime will be slower. If we relax the synchronization, then we can achieve better runtime with the sacrifice of some accuracy. Our goal is to relax the synchronization to some extent which will compromise a certain amount of accuracy but will reduce the runtime hence achieve better performance.

```
public class NoSyncWorker implements Runnable {
    private long workCount = 0;

    public void run() {
        doSomeWork();
    }

    public void doSomeWork() {
        //do some work
        workCount++;
    }

    public long getWorkCount() {
        return workCount;
    }
}
```

(a) Worker without Synchronization (*NoSyncWorker*)

```
public class SyncWorker implements Runnable {
    private long workCount = 0;

    public void run() {
        doSomeWork();
    }

    public synchronized void doSomeWork() {
        //do some work
        workCount++;
    }

    public long getWorkCount() {
        return workCount;
    }
}
```

(b) Worker with Synchronized Keyword (*SyncWorker*)

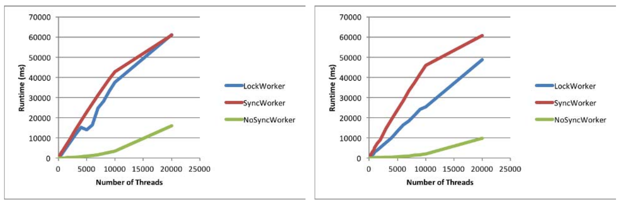
```
public class LockWorker implements Runnable {
    private long workCount = 0;
    private WriteLock lock =
        new ReentrantReadWriteLock().writeLock();

    public void run() {
        doSomeWork();
    }

    public void doSomeWork() {
        lock.lock();
        //do some work
        workCount++;
        lock.unlock();
    }

    public long getWorkCount() {
        return workCount;
    }
}
```

(c) Worker with Lock (*LockWorker*)

Figure 1. Different multithreaded workers

We choose lock over synchronized keyword for our approximate computing framework due to the following reasons:

- Though both synchronized keyword and lock ensures full synchronization, Figure 2 clearly shows that synchronized keyword needs more runtime than lock for larger number of threads in Java 8. So using lock over synchronized keyword is more logical for performance.
- It is not possible to tweak synchronized keyword for relaxed synchronization. It will always provide full



(a) Java 7 **(b)** Java 8

Figure 2. Performance comparsion of different multi-threaded workers

synchronization.

- The ReentrantLock of Java concurrency utilities framework consists tryLock() as shown in Figure 3 which provides a non-blocking attempt to acquire a lock that can be interrupted and time out. Our framework for approximate computing in Java is based on this tryLock of ReentrantLock.

```java
public class TryLockWorker implements Runnable {
    private long workCount = 0;
    private WriteLock lock =
        new ReentrantReadWriteLock().writeLock();

    public void run() {
        doSomeWork();
    }

    public void doSomeWork() {
        if (lock.tryLock(VERY_LARGE_TIME)) {
            //do some work
            workCount++;
            lock.unlock();
        }
    }

    public long getWorkCount() {
        return workCount;
    }
}
```

Figure 3. TryLock Worker

D. Interesting Findings

We encountered some interesting findings related to thread creation and synchronization. The findings are listed below:

Unstable behaviour with ReentrantLock. tryLock() and Thread.sleep(): We tried to use Thread.sleep() to forcefully suspend threads for some specific amount of time. Using ReentrantLock.tryLock() and Thread.sleep() together leads to an unstable behavior related with the number of times the function is actually called. To illustrate, if the code snippet of Figure 3 with Thread.sleep() inside tryLock() is called for 100 times this will actually be executed for 80-90 times.

Possible Reasons: From the observation, we can identify one probable reason. While using synchronized keyword, waiting for an entry to a busy critical section sends the thread into the waiting queue. If we use wait() in that critical section, then it conflicts with the waiting queue. Hence calling notify() or notifyAll() allows all the threads into the critical section. This is properly implemented by checking the entry condition in a tight loop instead of a simple condition. We assume the similar thing happens with ReentrantLock.tryLock(). In tryLock() the threads are sent to sleeping queue instead of waiting queue. So, when we explicitly use Thread.sleep() for synthetic thread suspension these threads conflict with the sleeping queue.

Unstable behaviour with ReentrantLock.tryLock() and creation of new threads: We predicted that extending Thread class instead of using ExecutorService to create new thread may resolve the above problem. But if we implement Runnable or extend Thread class for creating new thread, ReentrantLock.tryLock() doesn't work at all, it behaves like default lock() method.

Possible Reasons: We think this may be a bug and we will report it.

IV. Design

Our goal is to design and develop a framework which will allow any data structure to operate approximately. We choose to use relaxed synchronization to achieve this goal. We introduce approximate lock, an adaptive tryLock for approximate computation. When tryLock is used, a thread does not wait to acquire that lock if the tryLock wait time has exceeded. Figure 4 represents our *AdaptiveWorker* where each thread waits for a given time of the tryLock. If the waiting time exceeds, the thread skips acquiring the lock. We adaptively manipulate that wait time for our approximate lock to achieve user-specific error bound. We see from the analysis that *NoSyncWorker* provides faster runtime but very low accuracy but both *SyncWorker* and *LockWorker* provides perfect accuracy but compromises runtime efficiency. Our approximate *AdaptiveWorker* lies in between that achieves better runtime by compromising accuracy.

A. Design Parameters

The main parameters of our design are described below:

1) Initial Wait Time: At the beginning, we fix the initial wait time t_0 which will be adaptively changed. We perform several experiments to find out the optimal value of initial wait time. The performance optimal value of the initial wait time needs to be a multiple of the time t_w required to do the work in the critical section. So the initial wait time t_0 can be defined as follows:

$$t_0 = t_w \times c_a \qquad (1)$$

Here, c_a is a constant. We perform different experiments with various c_a to acquire the performance optimal value of c_a.

```java
public class AdaptiveWorker implements Runnable {
    private long initialWorkTime;
    private long adaptiveTime;
    private long workCount = 0;
    private WriteLock lock =
        new ReentrantReadWriteLock().writeLock();

    AdaptiveWorker() {
        initialWorkTime = calculateWorkTime();
        adaptiveTime = SOME_CONSTANT * initialWorkTime;
    }

    public void run() {
        doSomeWork();
    }

    public void doSomeWork() {
        if (lock.tryLock(adaptiveTime)) {
            //do some work
            workCount++;
            updateAdaptiveTime();
            lock.unlock();
        }
    }

    public long getWorkCount() {
        return workCount;
    }

    void updateAdaptiveTime(){
        long error = calculateError();
        if(error > tolerableError){
            adaptiveTime += stepTime;
        } else {
            adaptiveTime -= stepTime;
        }
    }

    long calculateWorkTime() {
        long startTime = Time.now();
        //do some work
        long endTime = Time.now();
        return endTime - startTime;
    }
}
```

Figure 4. *AdaptiveWorker*

2) Step Size: To make wait time of our approximate lock adaptive, we need to have a variable step size δt. The wait time will increase by the amount of step size when the current error is greater than the user-defined error and will decrease vice versa. This step size is the most important design parameters to achieve good performance. A wrong step size may not reach the user defined error ever, so a careful selection of the step size is important. Our experiments guide us that this step size depends on the time needed to do the work t_w and the number of threads waiting w_t.

$$\delta t = t_w * w_t \tag{2}$$

3) Error Threshold: The user will define an error threshold E_t. We are calculating the current error E by the difference of how many times the work to be done is tried and how many times the work is actually done. This error E works as the controlling factor for the adaptive change of the wait time. If E is greater than E_t then the wait time t is decreased by the step size δt. If E is less than E_t then t is increased by the step size δt. In the other hand, if E equals E_t then t remains unchanged.

$$t = t \pm \delta t \tag{3}$$

4) Sampling Frequency: Calculating error and updating wait time incurs a cost at runtime. So the frequency of this update needs to be controlled. If we execute this update in each iteration, then the cost of this update will neutralize the effect of approximate computation. These updates need to be executed after a certain number of iteration which is an important parameter of our design. If this number is too large then the effect of approximate computation will be too nominal. But if it is too small, the effect of approximate computation will be neutralized by the actual time needed for these updates.

B. Main Algorithm

The following pseudocode gives an overview of the adaptive algorithm of our design. If a thread cannot acquire the lock within time t then it will not do the work, hence current error will increase. We control the user defined error threshold E_t by adaptively changing t.

```
1:  procedure WORK-COMPUTE-APPROXIMATE
2:      if lock.tryLock(t) then
3:          if numberOfIteration%sampleSize == 0 then
4:              $E \leftarrow$ getError()
5:              $w_t \leftarrow$ numberOfWaitingThreads()
6:              $t_w \leftarrow$ timeNeededForWork
7:              $\delta t = t_w * w_t$
8:              if $E > E_t$ then
9:                  $t \leftarrow t + \delta t$
10:             else if  then
11:                 $t \leftarrow t - \delta t$
12:             end if
13:         end if
14:         // do some work
15:         lock.unlock()
16:     end if
17: end procedure
```

V. Methodology

We use Ubuntu 14.04.1 LTS server distribution and a 64-bit (x86_64) 3.8.0-29 Linux kernel. We report both analysis and performance results on a 3.4 GHz, 22 nm Core i7-4770 Haswell with 4 cores and 2-way SMT. The two hardware threads on each core share a 32 KB L1 instruction cache, 32 KB L1 data cache, and 256 KB L2 cache. All four cores share a single 8 MB last level cache. A dual-channel memory controller is integrated into the CPU with 8 GB of DDR3-1066 memory.

VI. Results

We implement our designed framework and performed experiments to verify its correctness as well as performance. Our design can be integrated with most of the existing data structures. For our experiment, we use single linked list, a simple and widely used data structure. In a single linked list contents are stored in the list as nodes and a pointer or reference points to the next node in the list. Single linked list generally supports three major functions — add, delete and search. We execute this single linked list for a variable number of threads where each thread adds/deletes a variable number of nodes to/from the list. For simplicity, we only consider adding nodes in our experiment. This add function is the *do some work* of our approximate *AdaptiveWorker* in Figure 4. We can measure the number of nodes different threads try to add to the list. The length of the list indicates the number of nodes successfully added. Using this information, we can calculate the current error. We perform experiments in various scenarios to check the accuracy and runtime of our implementation. We vary the number of threads from 25 to 10000. We vary the value of the following design parameters to check their effect on the program: a) Constant c_a of initial wait time, b) sampling frequency, and c) error threshold. Finally, we compare the result of our implemented framework with synchronized keyword and lock.

A. Varying c_a

We plot runtime vs c_a for a various number of threads in Figure 5, given other parameters are constant. We can see that for very low value of c_a our framework fails to work as it fails to adjust to the required wait time. From Figure 5 we can conclude that any value between 20 to 50 is a valid value for c_a.

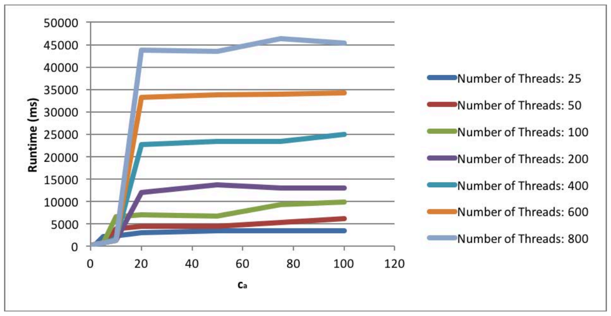


Figure 5. Varying c_a with number of threads

B. Varying Sampling Frequency

We plot runtime vs sampling frequency for a various number of threads in Figure 6, given other parameters are constant. We can see that for larger sampling frequency runtime decreases but error controlling fails. From Figure 6, we can infer that $10-20$ is a reasonable choice for sampling frequency.

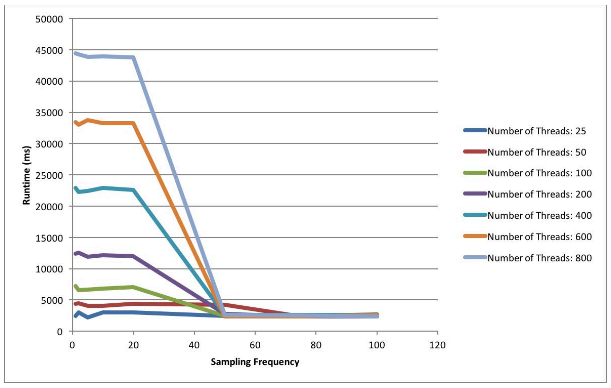


Figure 6. Varying sampling frequency with number of threads

C. Varying Error Threshold

We plot runtime vs error threshold for a various number of threads in Figure 7, given other parameters are constant. We can see that with the increase of error threshold, the runtime decreases. Because by increasing the threshold we are allowing more errors and that allows less number of nodes to be added to the list. It validates our assumption that by compromising a certain amount of accuracy we can decrease the runtime.

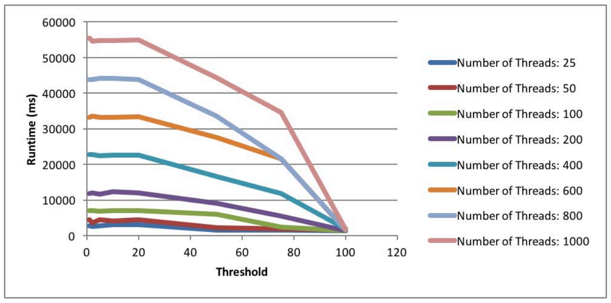


Figure 7. Varying error threshold with number of threads

D. Performance Measurement with Runtime

We plot runtime vs number of threads for *SyncWorker*, *LockWorker*, and our *AdaptiveWorker* in Figure 8. We can clearly see that our *AdaptiveWorker* with approximate lock achieve better performance than the other two by sacrificing accuracy in a controlled but acceptable way.

The insights and results suggest that our adaptive framework based on relaxed synchronization may be fruitful for approximate computing when the user can accept computation results with some tolerable error but with faster runtime.

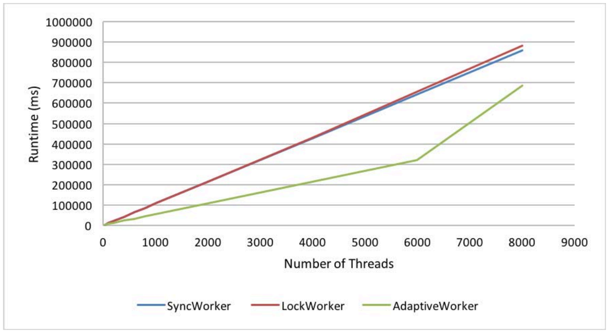


Figure 8. Final performance result

VII. Future Work and Conclusion

We propose a framework for approximate computing in Java with a specific focus on data structures. Currently, our framework supports single linked list data structure. We plan to extend it for data structures like doubly linked list, stack, queue etc. Moreover, we are developing a Java library which will contain these approximate data structures so that the programmers can use these easily and efficiently. Our next step will be make our approximate lock based data structures more independent. To elaborate, we would like the compiler to decide whether to use the precise data structures or the approximate ones based on the characteristics of the program. This benefits the programmers by eliminating the hassle of deciding whether to use approximate computation for a certain program.

Approximate computing opens a new door of possibilities in the era of modern computing. Relaxed synchronization in Java can provide approximate results but hasn't yet received much attention of the researchers. In Java, built in synchronized keyword achieves full synchronization but it costs a lot in performance. In this paper, we propose a framework that reduces the synchronization overhead of a program that can accept approximate results. Though it cannot achieve the full accuracy, we are able to reduce the time needed to execute the program. As a result, our framework for approximate computing provides better performance. Engraving approximate computing in Java will facilitate the programmers to engulf the flavor of approximation without taking much hassle.

Acknowledgment

This work is performed as a part of the undergraduate thesis in the Department of Computer Science and Engineering of Bangladesh University of Engineering and Technology (BUET). Our special thanks to BUET for providing such a good environment for research activities in the field of Computer Science and Engineering.

References

[1] M. Rinard, "Probabilistic accuracy bounds for fault-tolerant computations that discard tasks," in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 324–334.

[2] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *ACM Sigplan Notices*, vol. 45, no. 6. ACM, 2010, pp. 198–209.

[3] Q. Zhang, F. Yuan, R. Ye, and Q. Xu, "Approxit: An approximate computing framework for iterative methods," in *Design Automation Conference (DAC), 2014 51st ACM/EDAC/IEEE*. IEEE, 2014, pp. 1–6.

[4] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A programmer-guided compiler framework for practical approximate computing," *University of Washington Technical Report UW-CSE-15-01*, vol. 1, 2015.

[5] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," 2009.

[6] G. Pekhimenko, D. Koutra, and K. Qian, "Approximate computing: Application analysis and hardware design."

[7] M. C. Rinard, "Unsynchronized techniques for approximate parallel computing," in *RACES Workshop*, 2012.

[8] 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*. ACM, 2012, pp. 41–50.

[9] J. Gustedt and E. Jeanvoine, "Relaxed synchronization with ordered read-write locks," in *Euro-Par 2011: Parallel Processing Workshops*. Springer, 2011, pp. 387–397.

[10] M. Paleczny, C. Vick, and C. Click, "The java hotspot tm server compiler," in *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium-Volume 1*. USENIX Association, 2001, pp. 1–1.

[11] R. R. Newton, P. P. Fogg, and A. Varamesh, "Adaptive lock-free maps: purely-functional to scalable," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2015, pp. 218–229.