Quantifying and Reducing Execution Variance in STM via Model Driven Commit Optimization

Girish Mururu
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
girishmururu@gatech.edu

Ada Gavrilovska
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
ada@cc.gatech.edu

Santosh Pande
School of Computer Science
Georgia Institute of Technology
Atlanta, USA
santosh@cc.gatech.edu

***Abstract*—Simplified parallel programming coupled with an ability to express speculative computation is realized with Software Transactional Memory (STM). Although STMs are gaining popularity because of significant improvements in parallel performance, they exhibit enormous variation in transaction execution with non-repeatable performance behavior which is unacceptable in many application domains, especially in which frame rates and responsiveness should be predictable.In other domains reproducible transactional behavior helps towards system provisioning. Thus, reducing execution variance in STM is an important performance goal that has been mostly overlooked. In this work, we minimize the variance in execution time of threads in STM by reducing non-determinism exhibited due to speculation. We define the state of STM, and we use it to first quantify non-determinism and then generate an automaton that models the execution behavior of threads in STM. We finally use the state automaton to guide the STM to avoid non-predictable transactional behavior thus reducing non-determinism in rollbacks which in turn results in reduction in variance. We observed average reduction of variance in execution time of threads up to 74% in 16 cores and 53% in 8 cores by reducing non-determinism up to 24% in 16 cores and 44% in 8 cores, respectively, on STAMP benchmark suite while experiencing average slowdown of 4.8% in 8 cores and 19.2% in 16 cores. We also reduced the variance in frame rate by maximum of 65% on a version of real world game Quake3 without degradation in timing.**

***Index Terms*—Software Transactional Memory (STM), Execution Variance, Quake3**

I. Introduction

The execution time varies significantly across runs for parallel programs. Even for sequential programs, execution varies because of several factors such as co-executing programs, context switches, architectural artifacts like cache-misses, branch miss-predictions, non-deterministic memory access latency. In addition to these factors, in parallel programs the variance can be more dramatic because of interference amongst threads, sharing of resources and scheduling decisions. For example, migrations of threads by schedulers and sharing of cache between multiple threads cause cache misses and timing issues [19]. The execution of a critical section in different thread-interleaving orders increase non-determinism and timing variance in execution of the application.

Many applications, especially responsiveness oriented and soft-real time applications require a certain amount of repeatability of timing behaviors and in some cases prescribe a loosely acceptable bound on the amount of variance in execution time that could be tolerated [13]. Such limits on execution variance is usually because of the requirements on acceptable bounds on frame rates or jitters, or response time limits that emanate due to application requirements. For example, many games or multimedia codecs operate within certain frame rates for a smooth user experience, and several streaming applications must obey bounds on frame-rates for meeting the buffer restrictions, and certain web queries must execute with user response time limits. Moreover, the design of some asynchronous systems often implicitly assume timing characteristics that should be met which otherwise leads to timeout and retry.

Usually, the design goal of the system that is sensitive to execution variance is to respond in a certain time bound so that deadline misses are minimum. The application can be tuned to meet deadlines, achieve frame-rates, bound jitters, and maintain throughput and responsiveness through the measure of *variability* in the execution time. Less variability and high repeatability are the two important performance metrics that guide the performance optimization of these applications. When such applications are developed using transactional memory (TM), a parallel programming paradigm free of locks, the variance increases because of speculative execution. TM provides a clean abstraction for efficiently writing bug free parallel programs [4] in comparison to the use of locks that adds complexity to parallelization–the intricacy of avoiding deadlocks, livelocks, lock convoying, and priority inversion, and the difficulty of debugging is a programmer's nightmare [14]–and because of a decade of intense research and optimization, TM is now an attractive solution with good performance coupled with high programmer productivity for scalable workloads that are encountered in multi-media, gaming, and streaming computations.

Speculative execution with uncontrolled aborts throws a wrench in the desires of bounded variation expected by variance sensitive applications. For example, in SynQuake, a version of Quake3 game, the variance in frame rate processing time was observed to be 2.17 seconds, a value at which game play will be ridden with jitters. Also, applications in STAMP benchmark suite varied in execution by as much as 8 seconds in k-means. [1]

[1] Artifact available at: https://doi.org/10.5281/zenodo.1486191

The requirements in these modern workloads motivate this work that deals with minimizing the execution variance of Software Transactional Memory (STM) based parallel programs with minimal performance loss.

A. *Causes of Variance in STM*

In STM, several threads can simultaneously execute the code within a transactional section during which the STM logs the reads and writes to shared data and checks for conflicts before commit which otherwise is aborted and retried. Therefore, a thread can abort a number of times before finally committing a transaction. The aborts are not bounded, and a certain thread can abort distinct number of times while executing the same transaction under identical conditions due to non-determinism. That is, because of highly non-deterministic thread interleaving and architectural artifacts the number of aborts encountered by a thread can dramatically fluctuate. The overhead of aborts can be significant, and the execution variance compared to single thread is dramatic because of the number of aborts that could occur before commit [2].

The maximum non-determinism captured in terms of different commit orders, distinct order of threads completing a simultaneous transaction, in a n threaded application with no roll-backs is $n!$. For example, in four threaded application a maximum of only 24 distinct execution sequence of locked critical sections are possible without roll-backs. However, with roll-backs in STM, any upper bound on non-determinism disintegrates because of unbounded aborts. The commit order will now consist of threads that participated in the transaction but did not commit due to conflicts. Such threads retry and participate again in the execution of the critical section thus creating infinite possibilities in the execution order unlike lock based programs in which the variance of a thread can be bounded to the worst case sequentiality of a thread acquiring a lock first to acquiring the lock at last.

After seeing that the number of aborts and their fluctuations lead to high variance in execution time, one can aim at bounding the collective number of aborts of a given thread and at reducing variance in the number of rollbacks as a solution for reducing STM execution variance. A potential approach can be the one that prioritizes a thread after a certain number of aborts by assigning a commit priority. However, biasing STM in such a manner can sacrifice the essence of STM execution, i.e. speculation and fairness. In this work, we focus on devising a global solution to the problem of minimizing execution variances across *all* concurrent threads. This problem is more complex than application level variance reduction and cannot be characterized and solved by approaches that are based on roll-back reduction or that allow bounded aborts or that target reduction in the variances of roll-backs because of the nature of transactional processing–the number of concurrent transactions can vary with time. For example, in a scenario in which one of the threads committed by aborting few other threads that now retry the transaction, a new thread can join this transactional race, and so the aborts, the commit orders, and the demands to minimize the variance will vary in this new transaction environment thus changing the effective threads that must commit to reduce overall non-determinism.

In order to capture the pattern of execution that does not neglect speculation, one can envision a model based on a probabilistic automaton that captures the state of concurrency of threads and determines the most common commit paths emanating from each state. Guiding the execution along only such high probability paths can lead to less non-deterministic, globally minimal aborts with some degradation without sacrificing speculation. In essence, the model captures the combined effects of rollbacks and new concurrent transactions proposed earlier in work [17]. On STAMPS benchmark suite, guided execution reduced thread timing variance up to 74% in 16 cores and up to 53% in 8 cores with average slowdown of 4.8% in 8 threads and 19.2% ($\approx 50\%$ slowdown in 2 benchmarks) in 16 threads over parallel versions. All the threads in each of the application experienced similar reduction in variance, an empirical evidence that fairness was not jeopardized. In addition, we also reduced the variance in frame rate by up to 65% on a version of a real world gaming application, Quake3, without degradation in timing.

II. Methodology

STMs provide options of eager and lazy conflict detection. Eager conflict detection continuously checks for conflicts to abort threads, while lazy detection waits until the commit of a transaction before checking for any conflicts thus reducing the total number of retries and aborts. Thus, demonstration of guided execution on eager detection mechanism is easily implied by the testimony on lazy conflict detection mechanism. Hence, TL2, an STM that employs lazy detection mechanism, is used in our work for guided execution. To generate the automaton required for guided execution, we define a set of terms that are outlined in this section followed by the description of the framework.

A. *TL2 STM*

TL2 (Transactional Locking II) [7] is a STM based on a combination of commit-time locking and a novel global version-clock based technique that together decide if a transaction should commit or abort. TL2 is an API based write-back STM that provides transactional semantics to an atomic region. At the start of a transaction, TL2 reads the global version clock and stores it in a variable rv, and within a transaction, every read from shared memory is logged, and writes to shared memory are stored in a buffer and read back from the buffer. After the execution of the transaction, TL2 attempts to commit the changes with its commit protocol during which TL2 acquires a lock to each of the memory location that was accessed and checks if any other threads have modified the locations read to or written by the transaction by comparing the current version number of the memory locations with the value in rv. If the locations have been modified, a conflict is raised and the transaction is aborted. If all the memory locations are unmodified, the changes from the buffer are written into the shared memory subsequently updating the version number of the memory location.

B. Definitions

We now introduce several key definitions.

Variance: The goal of the paper is to reduce the variance in the execution time of each thread. The execution time of a thread is the time required by the thread function to complete. The thread function consists of a number of transactions, and each such transaction can abort multiple times before committing thus varying the execution time. Hence, the execution time accounts for the number of rollbacks seen by the thread. In our work, we quantify the execution time variance as the standard deviation in the measured execution time readings. The standard deviation of execution time is calculated as follows:

$$s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2},$$

where x_i denotes each timing measured over multiple runs, and $\overline{x}$ is the mean value of the timings.

Thread Transactional State: At any point during the execution of a threaded application, the state of the transacting application is defined by the thread transactional state (TTS). If $T = \{t_1, t_2, ...T_n\}$ is a set of threads executing corresponding transactions in $C = \{c_1, c_2, ...c_n\}$, a thread transactional state s is the tuple that captures the outcome of the simultaneous execution of transactions such as $s = \{< c_j tj, c_k tk, ..c_l t_l >, c_i t_i\}$, in which t_i commits the transaction c_i causing threads in $T_a = \{t_j, .., t_l\}$ to abort their corresponding transactions in $C_a = \{c_j, .., c_l\}$. In other words, a state is a tuple consisting of the aborted threads IDs coupled together with their respective transaction IDs along with the concatenated thread ID and transaction ID of the one that committed for each commit. For example, let threads 1,2,3, and 4 execute concurrent transactions a,b,c, and d, respectively. If thread 4 committed the transaction d aborting other threads 1,2, and 3, then the state would be designated as $\{< a1b2c3 >, < d4 >\}$. Threads 1, 2, and 3 could retry transactions by themselves, or another thread could join them. In the latter case, the tuple could look like $\{< a1c2e5 >, < c3 >\}$ in which threads 1, 2, and 5 were aborted by thread 3. Whenever a single thread executes and commits a transaction, the tuple entails only the thread concatenated with the transaction. For example, $\{< c3 >\}$ denotes that thread 3 executed and committed a transaction c. The adequateness of this definition to capture the transaction states is explained in Section III.

Non-determinism: A deterministic transactional execution will always repeat the same sequence of events on repeated runs which is only possible when aborts are zero and commits are completely deterministic. If aborts are zero and commits are not deterministic, then non-determinism can be measured by the different permutations exhibited. However, when aborts exist, they add to the non-deterministic behavior of the application due to unbounded aborts. The thread transactional state captures behaviors of both abort and commit. Hence, the total number of distinct thread transactional states that are exercised by the execution of an application serves as the measure of non-determinism that exists in concurrent execution.i.e.

$$Non - determinism = |S|,$$

where $S = \{s_1, s_2, ...s_n\}$ is the set of distinct thread transactional states.

Thread State Automaton(TSA): During the execution of application, some threads commit, some abort, and new threads could potentially join the aborted ones in the race of transactions. By logging the transactions, we find the succession of thread transactional states. From this trace, we construct a finite state automaton that shows the transition of the system from one state to another with a certain probability. This automaton called the Thread State Automaton (TSA) pictures the flow of transactional events in an execution.

Transition Probability: The edges in the TSA depict transitions from one state to another with certain probability. That is, a set of states can be reached from the current state with corresponding probabilities. We first sample the frequency of a transition from one state to another and then calculate the transition probability by dividing the frequency of a particular transition by the sum of frequencies of all outbound transitions. Thus, the probability of a transition is

$$P(e_i) = \frac{f(e_i)}{\sum_{j=1}^{n} f(e_j)},$$

where f(e_i) is the transition frequency of a given outbound transition edge e_i in TSA.

C. Framework

The framework ,shown in Figure 1, consists of four main phases as follows:

Profile Execution: In this phase, the application is executed multiple times to gather the sequence of aborts and commits during the run cycle of the application. This sequence is called the *transaction sequence (Tseq)*. The STM is modified to capture all commits and the corresponding aborts, if any. Each such tuple forms a thread transactional state (TSS).

Model generation: The transaction sequence is parsed to group thread transactional states (TSS). Each state is related to its neighbors with the transition probabilities by constructing a Thread State Automaton (TSA) that models the application behavior. The details of model generation is discussed in Section III.

Model Analysis: To determine the utility of the model for reducing variance, the generated model is analyzed by the model analyzer. If the model contains too few states, or if the probability distribution for transition are approximately equal, then the model is unfit for optimizing variance because the bias required for guiding the execution in a low variance path simply does not exist. This behavior is explained in detail in Section IV. Model analyzer correctly predicted the inability of the model to optimize ssca2 in STAMP benchmark suite that had mostly equally probable states.

Guided Execution: The generated automaton that passed the analyzer provides a probabilistic model of execution. In general, from a given state, several execution possibilities could result;

some of them could occur with a much higher probability than others. High probability transitions in the automaton indicate most common cases of execution. The model is used by the modified STM to guide the application towards a subset of high probability transitions by restricting transitions that wander to low probable states thus reducing variance. This phase is explained in detail in Section V.

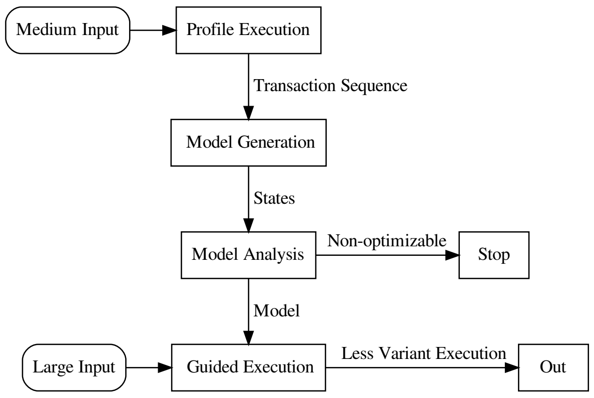


Fig. 1: Framework

III. Model Generation

The motivation behind model based approach is to capture the common cases of concurrent transaction execution of threads through aborts and commits. The model is then used for guiding the execution for which it needs to be detailed enough for high precision and simple enough for low overhead. We tried to generate a model using machine learning by involving all possible parameters in different stages of transaction execution such as logging, rollbacks, conflict detection. However, the model turned out complex and slowed down the guided execution significantly. We informally argue below why tracking the state of the concurrent transactions is sufficient rather than using a parametric model that involves tracking and using a large number of parameters.

A. Generating the State Model

To generate the thread state automaton (TSA), the application is profiled with a medium sized input and a modified TL2 library that captures the transaction sequence by logging the thread and transaction IDs of the commit and corresponding aborts, if any. The transaction sequence is parsed to generate a transition matrix representing the TSA as detailed in Algorithm 1. A TSA excerpt of kmeans is shown in Figure 3, in which the current state is $\{< a6 >, < b7 >\}$. From the definition of state, $< a6 >$ refers to the transaction a that was executed by thread 6 only to be aborted because of a conflict with the transaction b that was committed by thread 7. Similarly, the state $\{< b0 >\}$ refers to the transaction b that was executed and committed by thread 0 and so on. The states $\{< b0 >\}$, $\{< a1 >\}$, $\{< a2 >\}$, $\{< b3 >\}$, $\{< a4 >\}$, and $\{< a5 >\}$ are the most likely reachable from the current state $\{< a6 >, < b7 >\}$ because of their transition probabilities. The guided execution will probably follow one of these states. Since the model generation is carried out on a medium sized input and the runs are non-deterministic, not all possible states are captured by the model and any deadlock is resolved during model guidance. Models generated for different benchmarks and their properties are discussed in Section VII.

Algorithm 1 TSA generation Algorithm

```
1: procedure GENERATETSA
2:     TSeq = capture all commits and corresponding aborts
3:     TSS = tuple with commit and its abort, if any, in Tseq
4:     $D_S$ = set of neighboring TSS of unique S in Tseq
5: for each unique TSS S in Tseq:
6:     $N_i \leftarrow$ Number of transitions to $i \in D_S$
7:     $f_i \leftarrow \frac{N_i}{\sum_{k=1}^{n} N_k} \forall k \in D_S$
8: End for
```

IV. Model Analysis

Once a state model is built, the analysis phase validates the use of the model for reducing variance by analyzing the number of states and the transition probabilities of the model. During the execution, a state s_i can transition to several states in $S = \{s_j, s_{j_1} ... s_k\}$. However, if the transition from any such state s_i was always to only one corresponding unique state, the execution would become deterministic, and if the transition from any such state s_i was always contained within a constant set $S' = \{s_j, s_{j_1} ... s_{j_n}\}$, the amount of non-determinism and thus variance would be bounded by the distinct permutations of this set for all transactions. Guided execution tries to achieve an upper bound in variance by reducing the size of set S and converting the set to S', a constant subset of S consisting of only high probable states, by withholding the threads from deviating to a state that is not likely reachable in the model. The model analyzer verifies the difference that would result if the transactions were contained in S versus S' and determines if the model is useful in reducing variance.

A ratio of total number of transition states that are reachable in a guided manner from each state to that of the number of transition states that are reachable in the original unguided execution is the guidance metric that serves as a measure to determine if the model can be used for guided execution. The lower the metric, the better the benchmark is suited for guided execution. That is, if the difference between S and S' is higher, the more suited is the model for guided execution. The metric is obtained by traversing the TSA to find the cardinality of S and S' from all possible states. If the metric is above 50, we observed that most of the transition states in the model are high probability states.i.e.$|S| \approx |S'| \forall states$. In Table I, the guidance metric for ssca2 is very high at 72% and 57% for 8 and 16 threads, respectively. Hence, ssca2 is not suited for variance optimization through model driven commits which is further validated in our experiments in Section VII.

TABLE I: Model Analyzer guidance metric percentage (Lower is better)

Application	8 threads	16 threads
genome	34	40
intruder	32	36
kmeans	26	37
labyrinth	44	46
ssca2	72	57
vacation	31	28
yada	19	9

V. Guided Execution

The model that passes the validation phase is used for guided execution that limits the number of possible transitions to states that have lowest transition probability and is summarized in Figure 2. In guided execution, the STM allows transition from the current state S to a set D consisting of destination states with relatively high transition probabilities.

When a thread ,say $T1$, starts a transaction a, if $< aT1 >$ is part of any of the state tuples of the destination states in D, that is, $< aT1 >$ is either a commit or an abort in any of the destination states in D, then $T1$ is allowed to proceed, or else $< aT1 >$ is checked again to examine if it is a part of any of the state tuples of the destination states of the potentially changed current state because of a concurrent thread $T2$ committing a transaction b. $T1$ could find an execution path for continuing the transaction a from the new current state $U \in D$ as shown in Figure 2. If the current state does not change after k such retries, $< aT1 >$ is allowed to proceed to avoid deadlock and ensure progress. In essence, the framework holds back the thread transaction $< aT1 >$ that could commit with very low probability thereby avoiding repeated rollbacks, random commits that lead to low probability states, and jeopardizing other threads that are likely to commit. Because the STM executions are highly non-deterministic, training runs cannot possibly capture all execution states during model generation, so whenever a new state non-existent in the model is encountered, threads are allowed to continue so that the current state changes into a known state.

To demonstrate guided execution, we revisit the example in Figure 3, a thread transactional state excerpted from K-means executing on 8 cores, in which the current state seen is $\{< a6 >, < b7 >\}$ with likely destination states in set $D = \{< a1 >\}, \{< b3 >\}, \{< a2 >\}, \{< a4 >\}, \{< b0 >\}, \{< a5 >\}$ and low probability states $\{< c7 >\}$ and $\{< a6 >, < b5 >\}$. If $a6$ and $c7$, which are not part of tuples of high probability transition states, are the prevailing transactions, then execution can lead to distinct paths in two different scenarios as the transactions are blocked and retried. If during retry, any upcoming thread-transaction belonging to $b0$, $a1$, $a2$, $b3$, $a4$ or $a5$ will be allowed to proceed thus changing the current state. For example, if $b3$ committed, the current state would change to $\{< b3 >\}$ from which thread-transaction $c7$ will be allowed to continue. However, if within k retries none of the

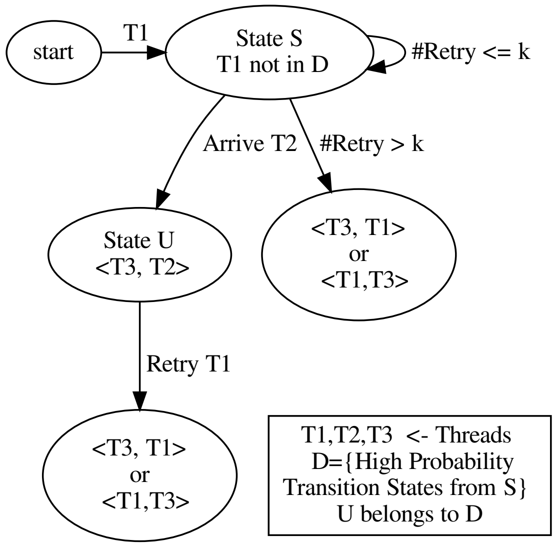


Fig. 2: Automaton for Guided Execution

current state changing transactions execute, to avoid deadlock transactions $a6$ and $c7$ will be allowed to progress.

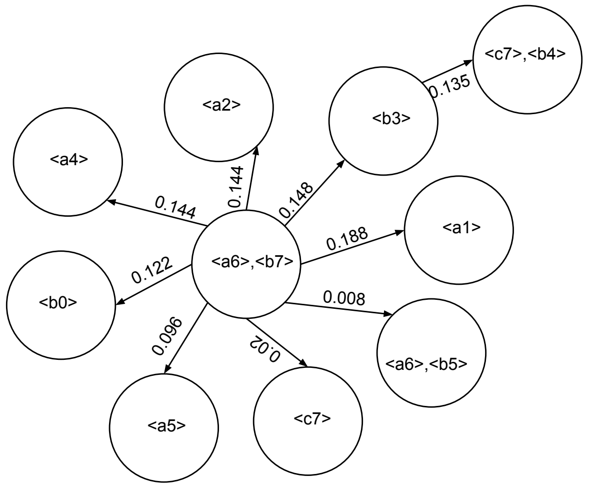


Fig. 3: State $\{< a6 >, < b7 >\}$ excerpt from kmeans

VI. Implementation

The destination states for each state in TSA is determined by a threshold transition probability. All state transitions with probability greater than the threshold is allowed to progress during guided execution. To determine the threshold probability, we first select the highest probability edge say with probability P_h. The threshold probability is then calculated as $P_h/Tfactor$.

$Tfactor$ is a knob to increase or decrease the amount of transitions possible at each state. A lower $Tfactor$ value allows for less number of transitions and restricts the STM, whereas a high $Tfactor$ value leads to path of low probability states with repeated aborts. By experimenting with $Tfactor$ values of between 1 to 10, we found that $Tfactor$ value of 4 strikes a balance.

We demonstrate our technique using TL2 and LibTM STMs. In TL2 benchmarks, each transaction is surrounded by APIs TM_BEGIN and TM_END that starts and ends a transaction, respectively. We changed TM_BEGIN to TM_BEGIN(ID) to account for the transaction ID that was statically numbered at source level by a script. We instrumented TX_start, TX_abort, TX_commit and TX_stop functions in the TL2 library to generate a bitwise transaction sequence. The benchmarks are first profiled to generate the sequence required for building the model. The size of the model built offline is on average 118KB in 8 cores and 1.3MB in 16 cores. However, during guided execution, the model is further cut down to exclude low-probability states and is stored in an efficient bitwise structure. A hash map is used to look-up the destination states in TSA thus minimizing slowdown.

VII. Experimental Results

TABLE II: Configuration of machines used for experiments

Features	8 core	16 core
Core count	8	16
CPU clock frequency	2.40 GHz	2.7 GHz
Sockets	2	2
Core/socket	4	8
LLC Cache	8 MB	24 MB
Memory	48 GB	48 GB
Page size	2MB	2MB

Because STMs are most effective and extract maximum parallelism on machines with a maximum of 16 cores [20], we experimented on 8 and 16 core machines with configurations listed in Table II and running Linux OS. To prevent interference of OS scheduling, we pinned the threads to cores. For our initial experiments, we used STAMP suite [1] that uses TL2 STM.

STAMP consists of eight benchmarks in which *Bayes*, a Bayesian network learning application, seg-faulted during experiments, an issue that has also been reported earlier in [24]. For each of the remaining seven benchmarks, models for 8 and 16 threads were generated separately using the medium sized input data sets. The models were then analyzed as explained in Section IV to determine whether model driven optimization of variance would be beneficial for the benchmark. Six out of seven benchmarks were passed by the analyzer as potential candidates for variance optimization, except for ssca2, whose model only consists few states that are likely reachable thus eliminating any scope for guidance.

The variance results, percentage reduction in standard deviation of execution time and comparison of the abort distribution in guided versus default executions, are reported for each thread in a benchmark. For each benchmark, we present percentage reduction in non-determinism and slowdown in the program execution over the original parallel version. STAMP benchmarks are not throughput oriented applications and the execution time of the benchmark is a metric close to throughput. For stable results, the readings were averaged over 20 runs and also the model was built with the transaction sequence generated from 20 runs with medium sized inputs. The number of states in each of the model is shown in Table III. We demonstrate that our technique reduces low probability states with high aborts through the improvement in the tail distribution of aborts as shown in Figures 5 and 7 for 8 and 16 threads, respectively. Because of space constraints, the tail distribution figures are shown for only few distinct threads. However, we list the average percentage improvement in tail distribution of aborts through a metric defined for each thread i in the application as follows:

$$tail_i = \sum_{j=0}^{n} (j)^2,$$

where j is the different number of aborts seen. To highlight the tail of the distribution, the distribution points are squared. Hence, the tail of the distribution that are of higher magnitude adds more weight to the metric. In other words, if the difference in the tail of the abort distribution is longer (high number of aborts with non zero frequency), the higher the value of the metric. The average percentage improvement of the tail distribution metric is shown in Table IV.

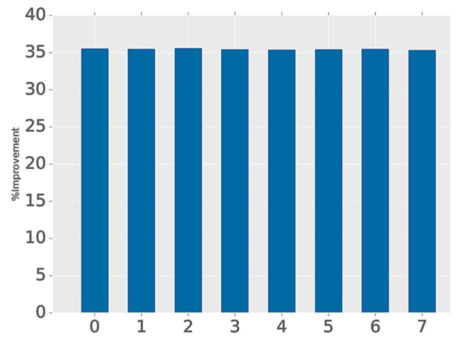
TABLE III: The number of states in the model of application

Application	8 threads	16 threads
genome	678	1555
intruder	71371	1352674
kmeans	3866	12689
labyrinth	445	797
ssca2	59	124
vacation	3781	15470
yada	27120	217606

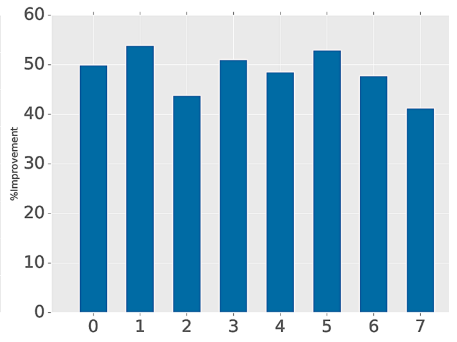
TABLE IV: Average percentage improvement in the tail distribution of aborts of all threads

Application	8 threads	16 threads
genome	76%	45%
intruder	82%	24%
kmeans	75%	40%
labyrinth	51%	11%
ssca2	0	0
vacation	26%	52%
yada	69%	29%

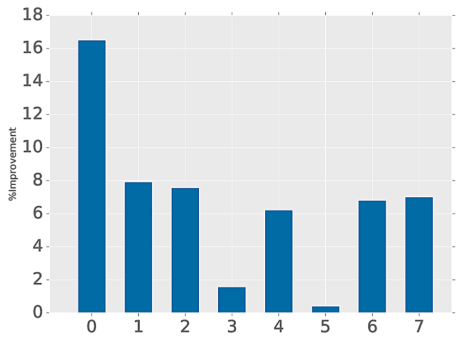
8Threads: Figure 4 shows the percentage reduction of variance of all 8threads in each benchmark. Variance in all the threads of every benchmark, except ssca2, is reduced between a range of 1 - 53%. Ssca2 with a total of 59 states in TSA and innately nearly zero aborts as shown in Figure 8b degrades by 8% in guided execution. The framework is an added overhead to the inherently less non-deterministic ssca2, because the number of aborts is unchanged between the default and guided executions as shown in Table IV. For



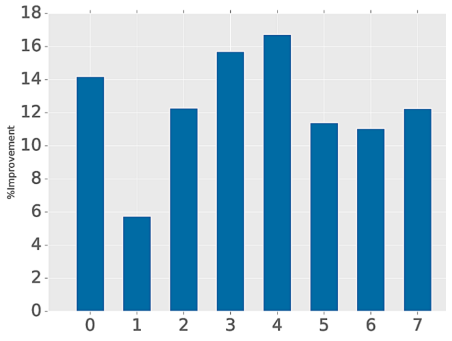
(a) genome



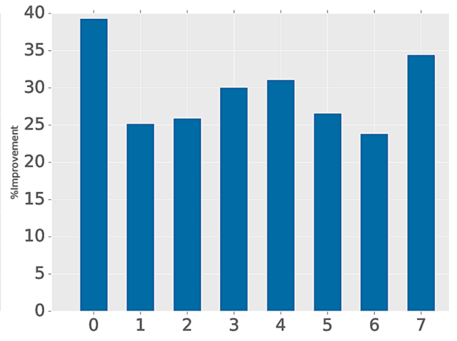
(b) intruder



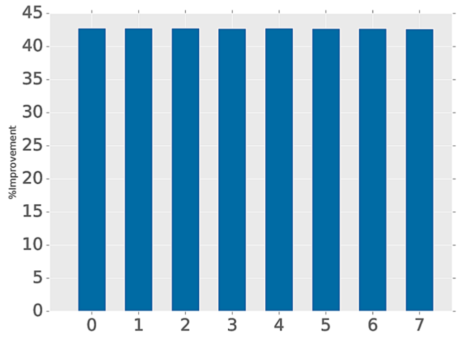
(c) kmeans



(d) labyrinth

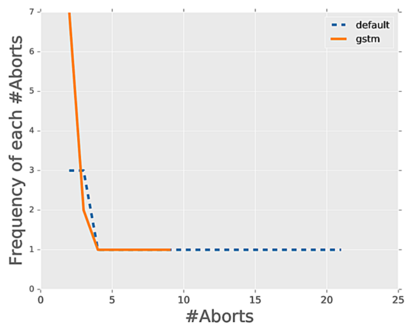


(e) vacation

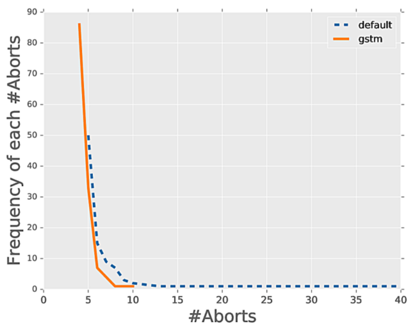


(f) yada

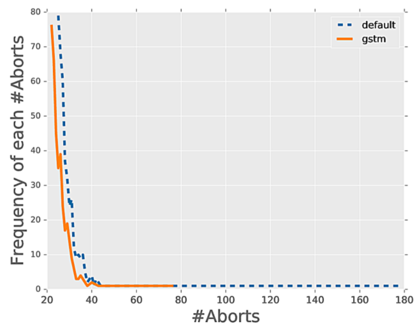
Fig. 4: Percentage Execution Time Variance Improvement of 8 threads per benchmark; y-axis denotes the percentage improvement and x-axis the threads



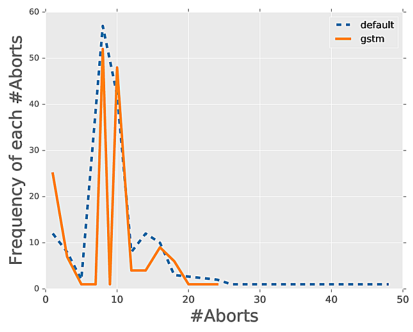
(a) genome thread 0



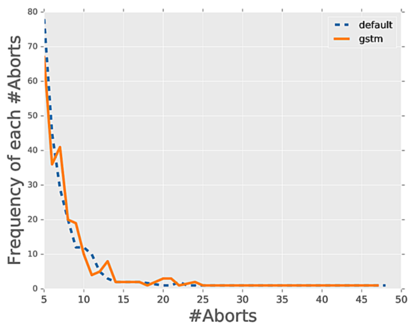
(b) intruder thread 1



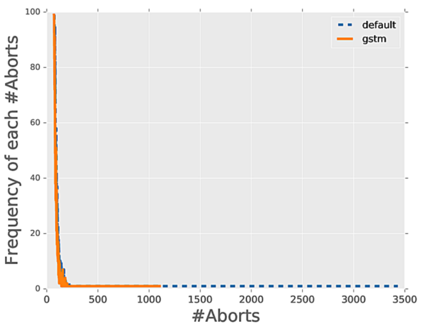
(c) kmeans thread 2



(d) labyrinth thread 3

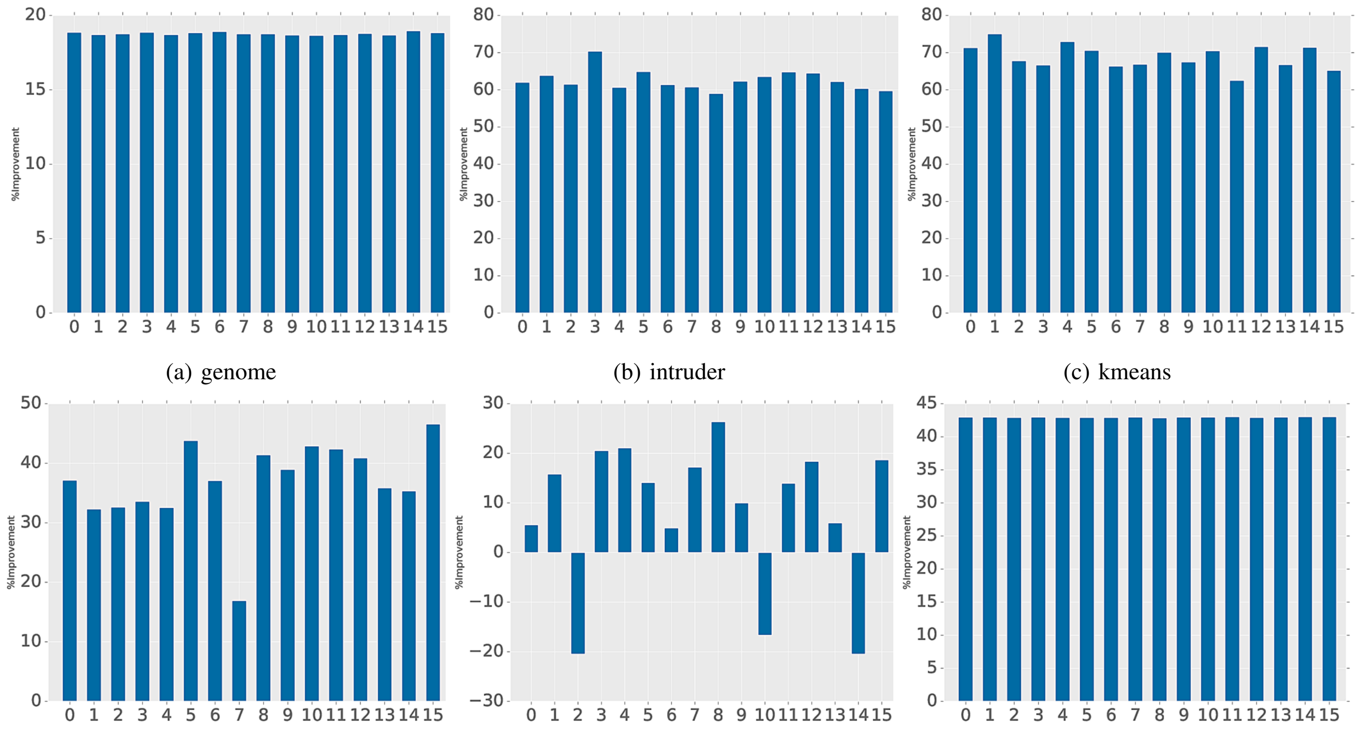


(e) vacation thread 7



(f) yada thread 6

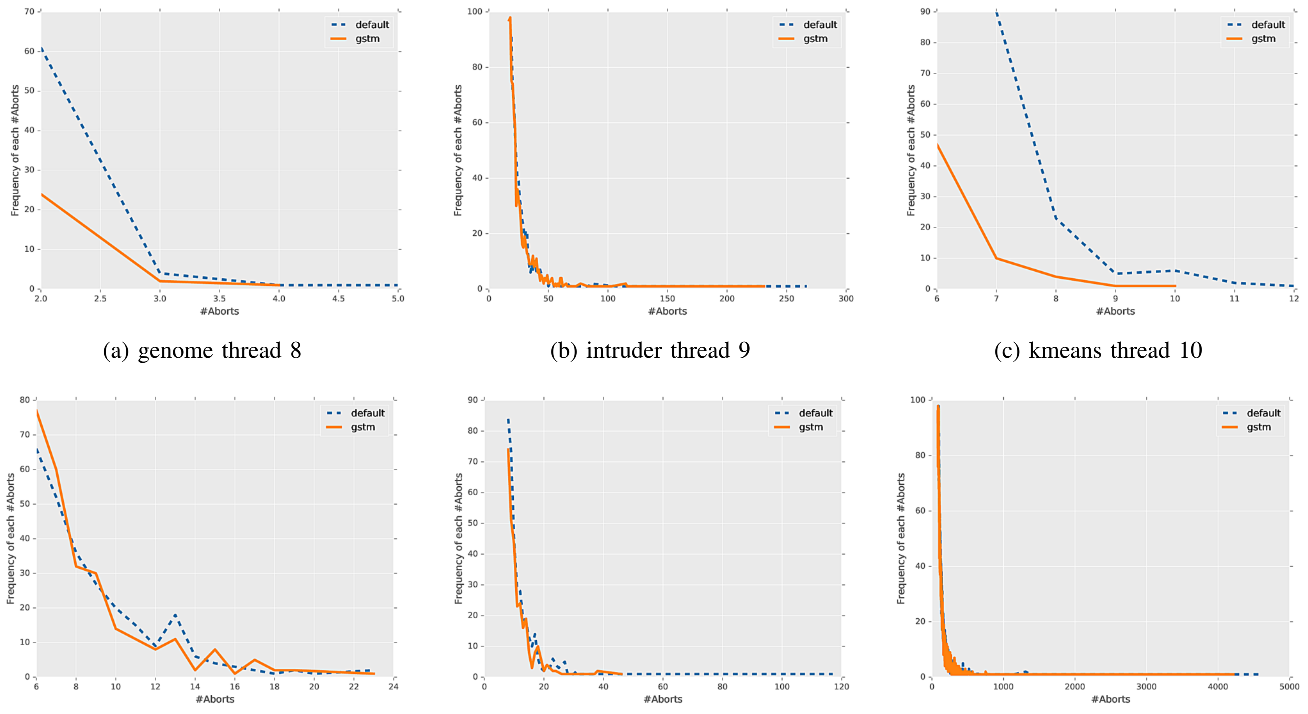
Fig. 5: Comparison of tail of the abort distribution with serially picked(0-6) threads of each benchmark. The dotted line corresponds to the default case and the solid line to our frameworky-axis denotes frequency of each abort



(a) genome (b) intruder (c) kmeans

(d) labyrinth (e) vacation (f) yada

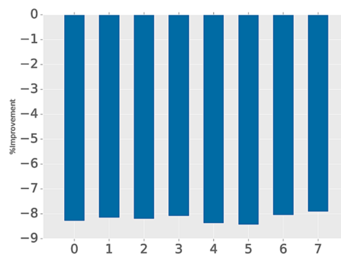
Fig. 6: Percentage Execution Time Variance Improvement of 16 threads per benchmark; y-axis denotes the percentage improvement and x-axis the threads



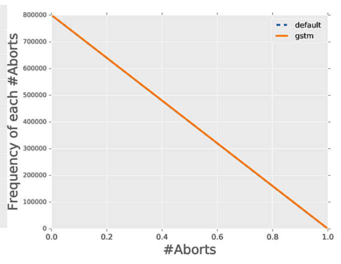
(a) genome thread 8 (b) intruder thread 9 (c) kmeans thread 10

(d) labyrinth thread 11 (e) vacation thread 13 (f) yada thread 14

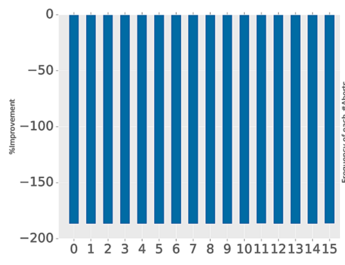
Fig. 7: Comparison of tail of the abort distribution with serially picked(8-14) threads of each benchmark. The dotted line corresponds to the default case and the solid line to our framework; y-axis denotes frequency of each abort



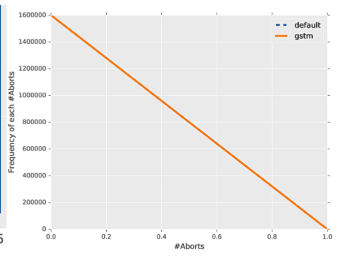
(a) ssca2 8threads



(b) ssca2 thread 4

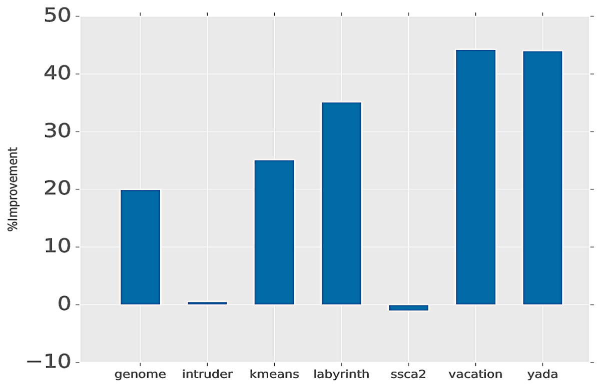


(c) ssca2 16threads

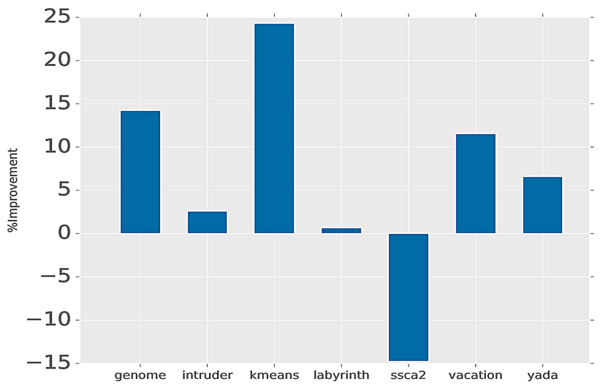


(d) ssca2 thread 12

Fig. 8: SSCA2 performance with guided execution. (a) and (c) denote percentage degradation (negative improvement) for 8 and 16threads, correspondingly. (b) and (c) shows the tail of abort distribution for thread 4 and thread 12 in case of 8 and 16 threads, respectively.

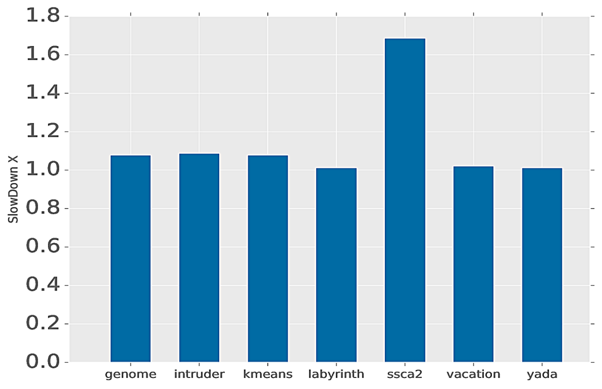


(a) Non-determinism 8threads

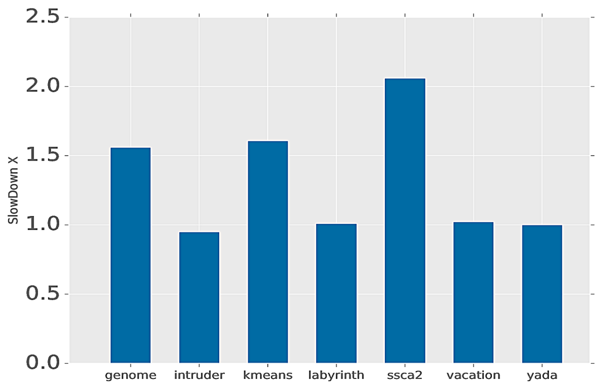


(b) Non-determinism 16threads

Fig. 9: Reduction in non-determinism of guided vs normal execution in 8 and 16 threads; the y-axis denotes the percentage reduction in Non-determinism



(a) 8threads



(b) 16threads

Fig. 10: Slowdown of guided vs normal execution in 8 and 16 threads;the y-axis denotes Slowdown(X)

all other benchmarks the tail of the distribution is cut down by the guided STM thus validating our claim that unbounded aborts are the primary cause of non-determinism in STM. Non-determinism is reduced by more than 40% at maximum as shown in Figure 9a. Although in intruder as many as 25000 states are reduced by guided STM in 16 threads, the percentage difference is less because of the large number of states. The benefits come at the cost of average slowdown of 3.5% as shown in Figure 10a.

16Threads: Figure 6 shows the percentage reduction of variance of all the 16 threads in each benchmark. Similar to 8 threads, ssca2 degrades by almost 186%. Along with the reasons as in case of 8 threads, the overhead is increased because of the increase in the number of equally likely reachable states. The number of aborts in ssca2 remains unchanged even in 16 threads. The intruder benchmark for 16 threads failed on every other run with large and medium sized inputs for both default and guided executions. Hence, experiments were performed with smaller sized input that improved execution timing by

5% as shown in Figure 10b. Although genome and k-means experienced a slowdown of 1.5 and 1.6 times in guided STM versus default, the standard deviation was reduced extensively.

Other benchmarks show similar improvement as in 8 threads except for vacation, a client/server travel reservation system that simulates flight booking management system. The threads in vacation behave as clients that are generated in a pseudo-random manner and perform several random operations on flight booking. This pattern is difficult to account for in the trained model of 16 threads compared to 8 threads. The 16 threads trained model is weaker than 8 threads in capturing the behavior and thus guiding towards less variant execution as seen in timing variance figures of vacation in Figure 4e and 6e,respectively. However, because of this randomness 16 threads experience aborts of very high frequency compared to 8 threads. Thus eliminating some of these high frequency aborts results in huge improvement in the tail distribution of aborts. Hence, tail distribution table IV shows an inverted reduction in abort distribution for 16 and 8 threads. This is substantiated by the higher absolute reduction in the number of states in 16 threads versus 8 threads.

Remarks: The benchmarks in STAMP perform uniform work per thread which does not allow for much bias among thread behavior and thus is a worst case scenario for guided execution. Also, any model that involves learning from training data requires a very good representative input set of data. The medium sized training set in STAMP is not usually a representative input thus weakening the model. To emphasize the benefits of our framework, we also use our framework for optimizing a real world game that is practical and provides representative inputs.

VIII. Optimizing a real world game in LibTM

SynQuake, a 2D version of the real world Quake 3 multiplayer game developed by [15], computes every aspect of gameplay but not the additional physics computation required for 3D such as gravity and explosions as in Quake3. The events with regards to additional physics computation likely occurs only on non-TM data and can hide the TM overhead. Thus, the computation included in SynQuake not only applies to Quake3 but also to most of the multiplayer games. SynQuake, compared to lock based implementation of the game, employs a fine grained consistency at object level eliminating false sharing and reducing contention time thus maintaining scalability and achieving speedup over the lock-based version of the game. SynQuake uses LibTM,a software transactional memory developed in [15] that provides four different conflict detection mechanisms varying from fully pessimistic, the read and write locks have to be acquired before accessing the data, to fully optimistic, the write locks are acquired at commit and reads proceed without requiring locks, and two conflict resolution mechanisms–wait-for-readers and abort-readers–to choose from. To be consistent with SynQuake paper and the TL2 STM above, we use fully-optimistic conflict detection and abort-readers conflict resolution in LibTM with guided STM ported for our experiments.

The experiments were conducted on 8 and 16 core machine with 1000 players using two input quests, specific areas in the map that attract players thus simulating a high interest area in the game play and the associated player movement patterns, namely, 4worst_case and 4moving, for training the model and two other quests– 4quadrants and 4center_spread6–for testing. The representative input quests was used in training 1000 frames, and other quests were used in testing 10000 frames on a map of 1024 by 1024, a set of parameters similar to that in the SynQuake paper. The guidance metric, shown in Table V, with lower value indicates huge scope for guiding the threads to avoid outlying behavior and unnecessary rollbacks.

TABLE V: SynQuake guidance metric (lower is better)

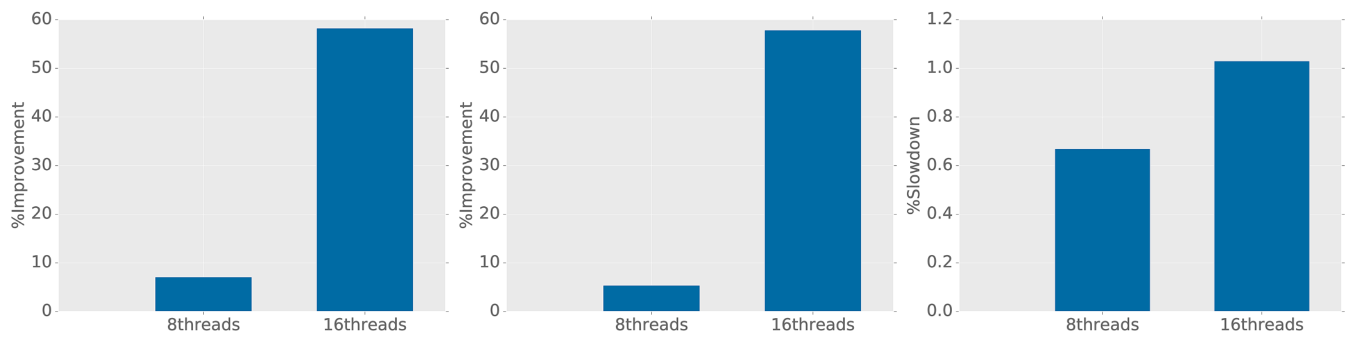
Application	8 threads	16 threads
SynQuake	22	19

The variance in processing frame rate is in order of seconds, a highest deviation of 2.17 seconds, a huge value that can cause jitters in game play. In case of 16 threads, this variance in frame rate was reduced by a maximum of 64.7% as shown in Figure 12a and number of aborts was also decreased by maximum of 57.86% as shown in figure 12b. The low guidance metric reasons the better performance of 16 threads compared to 8 threads, that is, compared to 8 threads the bias with 16 threads is much higher and thus larger scope for guidance. Because of the reduction in aborts, in 8 threads, SynQuake experiences a speedup of almost 35% with 4quadrants and 12% with 4center_spread6 as shown in Figures 11c and 12c, respectively. However, in 16 threads, shown in same figures, because of the increase in model data, speedup from reduced aborts is compensated by cache pressure exerted by the model. Because multiple client frames are handled by threads and executed within barriers, time variance per thread is not of significance and thus not reported.

IX. Related Work

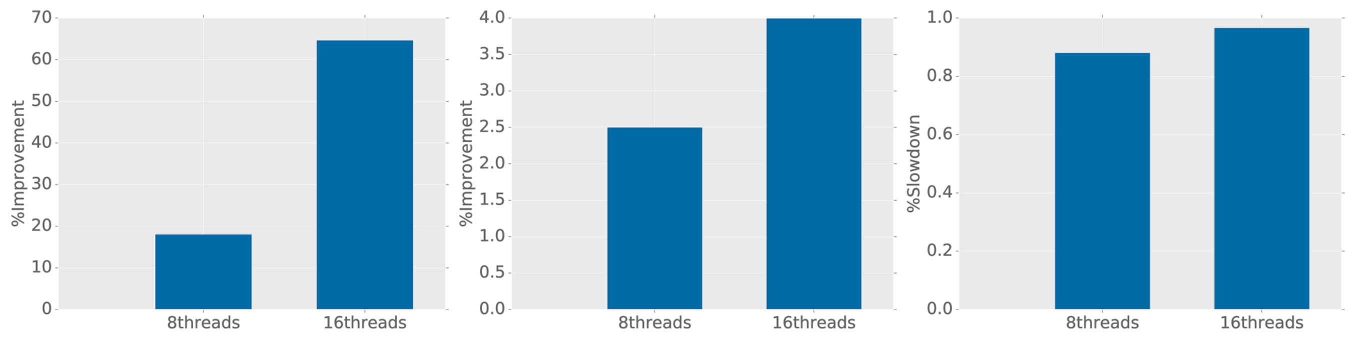
Transactional memory are of mainly three types–Software Transactional Memory (STM), Hardware Transactional Memory (HTM), and Hybrid models [5]. With constant advances in research, STM have been gaining traction in the community as an attractive alternative to using locks because of the significant improvements in their performance. Furthermore, STM systems provide a natural way of expressing speculative algorithms. There are several different STM designs. A few important design decisions involve: word-based vs. object-based, lock-based vs. non-blocking, write-through vs. write-back, or encounter-time locking vs. commit-time locking. Each of these can have significant impact on the transaction variance in different settings. Each have their own benefits and drawbacks and there is no clear winner.

Several authors have focused on Contention Managers (CMs) with the intent to reduce aborts, unlike this work that focuses on reducing variance through aborts. CMs withhold threads to reduce conflicts by allowing threads to proceed in a manner that results in higher throughput. For example, Polite [11] backs a thread off exponentially to let the competing thread



(a) FrameRate (b) Abort Ratio (c) Slowdown

Fig. 11: From a to c, % improvement in frame Rate variance, % reduction in Abort ratio, negative Slowdown(x) in 8 and 16 threads for quest $4quadrants$



(a) FrameRate (b) Abort Ratio (c) Slowdown

Fig. 12: From a to c, % improvement in frame Rate variance, % reduction in Abort ratio, negative Slowdown(x) in 8 and 16 threads for quest $4center\_spread6$

progress, Karma [22] prioritizes the thread that has accessed more transactional objects, Greedy [10] favors the thread with earliest start time. Thus, CMs clearly compromise one thread over another which only leads to higher variance. However, guided STM withholds threads only to follow a common execution path thus increasing the frequency of the most probable commit states with an aim to reduce variance of each thread.

A recent work DeSTM [21] guarantees completely deterministic execution of STM with the purpose of easier debugging of STM based applications. Another work in [23] allows transactions to proceed in an *irrevocable* mode such that they are not subjected to rollbacks due to conflicts. Deadline-aware-scheduling-for-Software-Transactional-Memory [16] tries to meet the deadline of certain transactions using irrevocable transactions. However, irrevocable transactions are mainly geared towards handling I/O operations and can adversely effect if used for reducing variance. Several authors in [3], [6], [9], [18] have targeted repeatability of non-STM program execution. Few other authors [8], [12], [13], [19] have tackled the problem of variance in non-STM systems. However, none of these techniques, unlike this work, deal with the unique problem of high variance and non-determinism that exists in STM due to extremely speculative execution.

X. Conclusion

In this work, we propose a model driven approach that optimizes variance in execution time for applications developed with STM. As STM is slowly being adopted for broader class of modern workloads, the need to minimize execution variance grows because of the highly non-deterministic behavior of STM due to speculative execution. Our approach generates a model and verifies its utility for optimizing variance. In STAMP, variance was minimized in six out of seven benchmarks with a maximum reduction of 74% in 16 cores and 53% in 8 cores by reducing non-determinism up to 24% in 16 cores and 44% in 8 cores with an average slowdown of 19% in 16 cores and 3.5% in 8 cores. While STAMP does not have inherent bias and medium sized inputs in STAMP did not capture the representative behavior, we successfully demonstrated our technique by reducing variance in frame rate of a version of Quake3 by a maximum of 65% in 16 cores and 18% in 8 cores with an average speedup of 1% in 16 cores and 35% in 8 cores.

Acknowledgment

The authors gratefully acknowledge [15] for providing us the source of LIBTM to demonstrate our work on Synquake.

Appendix

A. Abstract

The paper entails two parts of experiments. The first part evaluates stamps benchmark with TL2 STM and the second part evaluates synquake with LibTM. We provide the modified source of TL2 STM and stamps benchmarks along with the scripts required for execution and validating results. We provide one main execution script that compiles TL2 and stamps and executes the required option–profiling, guided, and default runs. We are not the authors of LibTM and synquake and are not in a position to disclose LibTM without which the synquake results cannot be verified. Other than the requirements for running the TL2 STM and stamps, which is a X86 machine, there are no requirements for running the software. However, for the experiments on the paper, we used an 8 core machine and a 16 core machine. The threads are pinned onto a specific core within the thread library of stamps (can be viewed in thread.c).

B. Artifact Check-list (meta-information)

- **Algorithm:** Yes.
- **Program:** We use stamps and are included. We have modified stamp-suite for including transaction id and for pinning the threads to cores.
- **Compilation:** We used gcc and we believe there is no restrictions.
- **Transformations:** No.
- **Binary:** No.
- **Data set:** The data sets are included and can be changed using an input to the script.
- **Run-time environment:** We used Linux and sudo access is not required.
- **Hardware:** X86 machines with 8 cores and 16 cores.
- **Run-time state:** Can be effected by state.
- **Execution:** The timing experiments are sensitive and can be affected. The threads are pinned already within the modified thread library of provided stamps. The profiling phase can take upto 10-30 minutes.
- **Metrics:** Execution time variance, execution time, non-determinism, abort distribution
- **Output:** The main execution script outputs a file with timings. Two scripts are provided to get difference in timings and non-determinism info is emitted. However, process-result can be checked to get abort-distribution information.
- **Experiments:** A single script, exec.sh, provided will prepare the environment and the inputs to the script will run different configurations required to produce the resutls.
- **How much disk space required (approximately)?:** 4-5 GB.
- **How much time is needed to prepare workflow (approximately)?:** 10 minutes
- **How much time is needed to complete experiments (approximately)?:** 1-4 hrs.
- **Publicly available?:** Yes.
- **Code/data licenses (if publicly available)?:** GNU license.
- **Workflow frameworks used?:** No.
- **Archived?:** Yes.

C. Description

1) How Delivered: Download package from https://doi.org/10.5281/zenodo.1486191

2) Hardware Dependencies: Other than the requirements for running the TL2 STM and stamps, which is a X86 machine, there are no requirements for running the software. However, for the experiments on the paper, we used an 8 core machine and a 16 core machine.

3) Software Dependencies: Linux OS should be the only software dependency.

4) Data Sets: Data sets are provided with the artifacts.

D. Installation

1) Download the file gstm_artifact.tar.gz
2) untar the file: tar -zxf gstm_artifact.tar.gz

E. Experiment Workflow

1) Goto Execute folder
2) All expts can be executed using the script exec.sh
3) For handy about do a tail -15 on exec.sh to know about inputs for the script
4) The script can be run as:
   ./exec.sh numthreads numruns option(lookabove) freq directoryName(RunName) where, different options are:
   - model - for using the model in guided execution (GSTM)
   - ND_mcmc - for generating the Non-deterministic data (secondary data) for guided STM
   - ND_only - for generating the non-deterministic data for default STM
   - mcmc_data - for generating the state model
   - default or orig - for default STM run

   freq - is the Tfactor and is usually set to 4.
   directoryName - is a name specified for the run (give it a name such as trial1) An optional size-of-data can be appended to the command different sizes-of-data are: large, medium, and small.
   Ignore stderr print messages like
   mv: cannot stat rdmp: No such file or directory
   mv: cannot stat abortsThread*: No such file or directory etc.

5) By default, the script uses medium size for training (generating the model (option= mcmc_data)) and small for testing.
6) However, the sizes can be varied as required by providing input to the size-of-data input argument.
7) Within the script the benches variable declares a list of benches. But some benchmarks like intruder fail on large sized input as mentioned in the results section. To run individual benchmark or remove few from list, edit exec.sh file to change benches variable in the script.
8) The final output numbers can be found in the file AvgSummary_"directoryName"_"option""freq"_"threads"_"num_runs" This file is necessary for final variance, timings, and non-determinism numbers for model and default runs. For mcmc_data runs (for generating model) this file is not important, although it is generated. A directory is created for each new bench -mark and mcmc_data option will generate the model, which is called state_data.
9) The script var_Percentagediff.py can be used to get percentage change in variance. Usage:./var_Percentagediff.py baseline_file GSTM_technique_file
10) Similarly, the runtime numbers can be obtained using ./avg_Percentagediff.py with same semantics.
11) The non-determinism numbers can be obtained by running the exec script with ND_mcmc option for guided STM and ND_only option for default STM. Each command will output a similar file as above (AvgSummary_file). However, the file in this case will have the benchmark name followed by number of states seen followed by the abort distribution. The abort distribution is in the format number-of-abort:frequency. For example, 0:700 implies that 700 times there were zero aborts and so on.
12) The experiments should maintain the following order:
    a) ./exec.sh numthreads numruns mcmc_data 4 directoryName
       The above run is for generating the model, which can be found as state_data file in the respective benchmark

directory created in the Execute folder after running the above command.

b) ./exec.sh numthreads numruns model 4 directoryName

c) ./exec.sh numthreads numruns default 4 directoryName

d) Compare the output files of the above two runs for timing results.

e) ./exec.sh numthreads numruns ND_mcmc 4 directoryName

f) ./exec.sh numthreads numruns ND_only 4 directoryName

g) Compare the output files of the above two runs for non-determinism results.

The size-of-data is optional and can be provided as additional argument.

F. Evaluation and Expected Result

The primary result in the paper is the variance in the benchmarks. The framework generates an output file with the variance numbers. We have provided a script to derive percentage change in timing variance and execution time. The Tfactor (FREQ) can influence the results and we have used a factor of 4. However, some machines might require a factor of 6. On re-running the experiments the variance results can vary because of inherent non-determinism. We expect that the variance in guided STM runs to be less than default STM. The number of states in guided STM must be fewer than default. Higher number of aborts will be present in default STM runs compared to guided STM runs, if they had same number of aborts, then you can also see that higher number-of-aborts were much frequent in case of default STM compared to guided STM.

References

[1] Chi Cao Minh and. STAMP: stanford transactional applications for multi-processing. In Christie et al. [4], pages 35–46.

[2] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Advanced concurrency control for transactional memory using transaction commit rate. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, pages 719–728, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[4] David Christie, Alan Lee, Onur Mutlu, and Benjamin G. Zorn, editors. *4th International Symposium on Workload Characterization (IISWC 2008), Seattle, Washington, USA, September 14-16, 2008*. IEEE Computer Society, 2008.

[5] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 336–346, New York, NY, USA, 2006. ACM.

[6] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: Deterministic shared memory multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 85–96, New York, NY, USA, 2009. ACM.

[7] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] Sherif F. Fahmy, Binoy Ravindran, and E. D. Jensen. On bounding response times under software transactional memory in distributed multiprocessor real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 688–693, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[9] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 367–384, New York, NY, USA, 2008. ACM.

[10] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 258–264, New York, NY, USA, 2005. ACM.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*, PODC '03, pages 92–101, New York, NY, USA, 2003. ACM.

[12] Christopher J. Hughes, Praful Kaul, Sarita V. Adve, Rohit Jain, Chanik Park, and Jayanth Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, ISCA '01, pages 254–265, New York, NY, USA, 2001. ACM.

[13] Tushar Kumar, Jaswanth Sreeram, Romain Cledat, and Santosh Pande. A profile-driven statistical analysis framework for the design optimization of soft real-time applications. In *The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*, ESEC-FSE companion '07, pages 529–532, New York, NY, USA, 2007. ACM.

[14] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[15] Daniel Lupei, Bogdan Simion, Don Pinto, Matthew Misler, Mihai Burcea, William Krick, and Cristiana Amza. Towards scalable and transparent parallelization of multiplayer games using transactional memory support. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 325–326, New York, NY, USA, 2010. ACM.

[16] W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller, and E. Riviere. Deadline-aware scheduling for software transactional memory. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 257–268, June 2011.

[17] Girish Mururu, Ada Gavrilovska, and Santosh Pande. Quantifying and reducing execution variance in STM via model driven commit optimization. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2018, Vienna, Austria, February 24-28, 2018*, pages 409–410, 2018.

[18] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[19] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread tranquilizer: Dynamically reducing performance variation. *ACM Trans. Archit. Code Optim.*, 8(4):46:1–46:21, January 2012.

[20] K. Ravichandran and S. Pande. F2c2-stm: Flux-based feedback-driven concurrency control for stms. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 927–938, May 2014.

[21] Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. Destm: Harnessing determinism in stms for application development. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 213–224, New York, NY, USA, 2014. ACM.

[22] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, New York, NY, USA, 2005. ACM.

[23] J. Sreeram and Santosh Pande. Hybrid transactions: Lock allocation and assignment for irrevocability. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1192–1203, May 2012.

[24] Gautam Upadhyaya, Samuel P. Midkiff, and Vijay S. Pai. Using data structure knowledge for efficient lock generation and strong atomicity. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 281–292, New York, NY, USA, 2010. ACM.