Modeling the Run-time Behavior of Transactional Memory

Zhengyu He and Bo Hong
School of Electrical and Computer Engineering
Georgia Institute of Technology
zhengyu.he,bohong@gatech.edu

Abstract—In this paper, we develop a queuing theory based analytical model to evaluate the performance of transactional memory. Based on the statistical characteristics observed on actual experiments, we model each transaction as a client requesting services from the computing system. Continuous time Markov chain is used to describe the start and completion (commit or abort) of the transactions. We analyze the mean transaction execution time to evaluate the performance of target transactional memory systems. Experimental results based on STAMP benchmarks show that our model can predict the performance of real transactional memory systems with an average error rate of 7.9%.

I. INTRODUCTION

Recently, transactional memory (TM) has emerged as a promising new paradigm for parallel programming on shared memory computers. It offers improved programmability over the traditional concurrency control mechanism of locks. With the trend shifting towards multi-core or multi-processor computing systems, many researchers are expecting TM to promote parallel programming for software development. Thus, more and more research efforts are being dedicated to the investigation of TM.

An attractive feature of TM is that programmers only need to locally consider about the shared data access and mark the code accordingly, and the underlying system will automatically ensure the correctness of concurrent executions. TM is thus able to significantly reduce the difficulties of parallel programming and debugging, and avoid the vulnerability to failures, and the likelihood of deadlocks.

Multiple factors affect the performance of TM based programs. When a transaction aborts because of a conflict, the computation that has been performed so far will be wasted. If the aborted transaction restarts, it may be aborted again, resulting in further waste. Intuitively, a longer transaction is likely to encounter more conflicts. The implementation overheads will inevitably prolong the length of transactions. Implementation overheads is actually closely associated with the ‘optimistic’ nature of TM systems: the immediate results must be buffered in either software or hardware so that a transaction can be rolled back in the case of a conflict [1]. Buffering takes time and resources, which may increase the contention level and thus intensify conflicts. The research community has been aware of the importance of both factors, which can be witnessed by a large amount of research efforts that are dedicated to explore TM design schemes (e.g. [2], [3]).

The objective of this paper is to provide a theoretical model that can reveal the relations between the performance and various key parameters of TM systems, including the length of transactions, the transaction arrival frequency, the number of checkpoints (the time points a transaction validate its read/write set), and the computing cost of transactions. In our model, we analyze the run-time behaviors of transactions for a given computer system, estimate the extra time that will be wasted due to conflicts, and obtain the expected execution time needed by the transactions. This approach differs from most existing studies where the focus is on the design and implementation of TMs, and the performance of the TM design is evaluated through simulations or actual executions by using benchmark suites such as STAMP [4]. Such empirical evaluation methods do provide very useful insight to TM studies, but are often unable to isolate the impact of an individual design option (because a typical TM system often integrates a collection of design options). We believe our analytical study would provide a useful tool for TM researches, especially for understanding the complex run-time behavior of transactional execution.

Our analytical model is based on Queuing Theory. In this model, each transaction is a client and the computing system acts as the server responding to the clients. Different from our previous results [5], [6] that focused on uniform transactions, this paper studies a general scenario in which multiple types of transactions may have partial conflicts. The correctness of our model is validated through extensive experiments using the STAMP benchmarks [4].

The rest of the paper is organized as follows: Section II summarizes the background and related works. Section III describes the target TM systems. Section IV presents our model and analysis. Section V presents the experimental results that validate the model and study the relation between TM performance and TM implementation overheads. Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK

The concept of transactional memory was borrowed from database systems to enforce atomicity and isolation for shared memory programming. The idea of providing hardware support for transactional memory originated in [7]
and has since been explored. STM was first developed with lock-freedom [8], and gradually evolved to obstruction-freedom [9] and now lock-based [10]–[13]. Schemes that mix hardware and software have also been explored [14], [15].

Various designs have been explored for TM systems. The three key aspects for TM designs are (1) conflict detection, (2) version management, and (3) conflict resolution [3]. Conflict detection decides when to examine the read/write-sets to detect conflicts, and the two popular design choices are eager or lazy. The eager option (e.g., in TinySTM [11] and EazyHTM [16]) attempts to detect conflict for every memory access. The lazy option (e.g., in TL2 [12] and TCC [17]) may delay the detection to the commit phase, which has been demonstrated to be able to avoid certain conflicts [3]. Version management handles the storage policy for permanent and transient data copies. Similarly, the policy can be either eager or lazy. In the TM systems with eager version management (e.g., TinySTM-WT and LogTM [18]), new data will take place of the old data in the memory and the old data will be logged. In the TM systems with lazy version management (e.g., TinySTM-WB, TL2 and VTM [19]), on the contrary, the old data is kept in place while the new data is logged. Conflict resolution manages the actions to be taken when a transaction encounters a conflict. Many options are available, such as blocking self, abort self, or abort others.

Most of the previous studies evaluate the performance of TM systems through experiments based on either simulations or actual executions. Such empirical evaluation methods have provided very useful insights to TM studies. On the contrary, there have been very few theoretical studies on the performance of STMs. Our previous analytical models [5], [6] adopted continuous time Markov chain (CTMC) and studied the mean transaction execution time. These models focused on the scenario of uniform transactions. The model developed by Heindl et al. [20] adopted discrete time Markov chain (DTMC) to investigate the impact of transaction conflicts. A “tagged” transaction is analyzed to represent the overall performance of TM systems. However, this model studies the expected number of retries before a transaction commits, which is not a direct measure of execution speed (because this model does not study the time needed to complete a retry, which may vary depending on the number of active transactions). Porter et al. [21] developed a tool called Syncnchar which models the workload performance of TM. This model statically estimates \( D_n \), the expected number of pair-wise conflicts assuming all \( n \) transactions execute simultaneously, and assumes that transactional execution of \( n \) threads will be slowed down by \( D_n \) times. This model, however, does not take into consideration that conflicts are dynamic and transactions with conflicting read/write-sets may not be executed simultaneously.

Compared with the existing analytical works, we study the dynamic run-time behaviors of transactions in this paper. Our analysis supports multiple types of transactions, including partially conflicting transactions.

### III. Target TM Systems

In this section, we specify the characteristics of the target TM systems that we consider, and define the notations for the rest of the paper.

Let \( N \) denote the number of threads in the system. And we assume each thread will be executed by one processor. Each thread is capable of executing each of the \( m \) transaction types. We assume that when a thread is executing a transaction, it does not start another one. We reserve the scenarios of nested transactions to our future study. When a thread is not executing any transactions, we call it a potential thread because it can potentially start a new transaction.

We use \( X_1, X_2, \cdots, X_m \) to denote the \( m \) types of transactions, where the proportion of \( X_i \) over all the instances of all types of transactions is \( p_i \). \( X_i \) has \( k_i \) checkpoints. Different TM designs will invoke different numbers of checkpoints for the same transaction. Each shared read/write has a chance to be a checkpoint. However, due to the reason of overheads, typically, a transaction will checkpoint after several shared reads/writes. When there are no conflicts (e.g., only one thread is running), the probability that a thread start a transaction within \( \Delta t \) time is \( \lambda \Delta t + o(\Delta t) \); and the probability that a thread reaches a checkpoint with \( \Delta t \) time is \( k_i \mu \Delta t + o(\Delta t) \). \( \lambda \) and \( \mu \) represent the arrival and service rates respectively when the transactions are executed sequentially. When multiple threads are concurrently executing transactions, \( \lambda \) and \( \mu \) may decrease if the shared computing resources (e.g., shared cache, interconnect, etc.) are saturated. We use \( C \) to denote the maximum rate at which the underlying computer system can supply the shared computing resources, and the computing cost of transaction \( X_i \) consumes these resources at a rate of \( c_i \). We assume the shared computing resources will be evenly distributed among threads. In such a scenario, the actual \( \lambda \) and \( \mu \) need to be corrected accordingly (details in Section IV-B).

When two transactions conflict, they must be accessing the same data and, more importantly, they should be running concurrently. Many complex conflicts exist including Read-after-Write (RAW), Write-after-Write (WAW), and Write-after-Read (WAR). To balance complexity against accuracy, we model the conflicts with \( p_{con} \), the probability that two transactions conflict (which may be detected at checkpoints as well as commit point). With this simplified conflict model, we can describe eager and lazy conflict detection strategies by considering the number of checkpoints. For instance, eager conflict detection will encounter more checkpoints so that it can have a higher probability to detect conflicts earlier. Upon the detection of a conflict, a transaction will abort itself. Such a strategy is adopted by many TM systems (e.g., TinySTM with suicide option). In our model, when two
transactions conflict, we assume the conflict will be detected by the transaction that has made less progress towards completion. This assumption is statistically reasonable - an ‘older’ transaction is expected to have accessed more shared variables than a ‘newer’ transaction, so the ‘newer’ transaction has a higher probability to detect a conflict. We reserve other conflict resolution strategies for our future study.

Figure 1. Illustration of transactional execution of the target TM systems

Fig. 1 illustrates an example of the target TM systems, $X_a$, $X_b$ and $X_c$ are three different transactions issued by two threads. $X_a$ illustrates that each transaction have a start, a commit and several ($k_a$ for $X_a$) checkpoints. Transaction $X_a$ finishes with rate $\mu$, and we assume each checkpoint will be finished at rate $k_a\mu$. When executing transaction $X_a$, Thread 1 cannot issue new transactions. But after finishing $X_a$, Thread 1 issues a new transaction $X_b$ with rate $\lambda$. If at the same time, Thread 2 issues a transaction $X_c$, it will detect the conflict with $X_b$ and abort $X_c$. $X_c$ will keep retrying until it commits.

IV. ANALYSIS OF THE TM SYSTEMS

A. Statistical Characteristics of Transactions

By profiling the STAMP benchmarks, we have observed Erlang distribution in the transaction completion time. In the experiments, the benchmarks were configured with recommended parameters in [4] and executed by 8 threads, where each transaction was executed 10,000 times. The results of four representative benchmark programs (other programs are similar) are shown in Fig. 2. Erlang distribution has 2 parameters: the shape $k$, which is a non-negative integer, and the rate $\mu$, which is a non-negative real number. Erlang distribution is closely related to exponential distribution in that the sum of $k$ independent exponential distributed variables with parameter $\mu$ is equal to an Erlang distributed variable with parameters $k$ and $\mu$. intruder is a special case in which $k = 1$ and the distribution reduces to an exponential distribution.

The above observation can be explained as follows: a transaction in general needs to pass several important time points during its execution (e.g. the start point, intermediate checkpoints where conflicts may be detected, and the commit point). We assume the probability $p$ of hitting such a time point is proportional to the amount of elapsed time $\Delta t$, i.e. $p = \mu \Delta t$ for some constant $\mu$. It can be easily shown that the time between two time points obeys exponential distribution with parameter $\mu$. When a transaction has $k$ such segments, the sum of the $k$ exponentially distributed segments is then equal to an Erlang distribution with parameter $k$ and $\mu$. The linear relation between $p$ and $\Delta t$ reflects the behavior of a typical program: given longer period of time, a program is expected to make more progress.

The execution of a transaction $X_i$ is therefore modeled as a sequence of events: Start, followed by $k_i - 1$ Checkpoints, and finally the Commit. For notational convenience, the commit point is considered as the $k_i^{th}$ checkpoint, where the time between two events obeys exponential distribution. When a transaction is aborted, it needs to repeat the sequence of events all over again until it can successfully commit.

B. Impact of Transactional Congestion

In practice, a computer system has finite computing resources. As defined in Section III, the overall shared computing resources has a maximum value $C$. When multiple threads are running concurrently in a system, the demanded amount of computing resources may exceed the system capacity and thus cause congestion. We assume every thread will be treated fairly in sharing the computing resources. We
use $R$ to denote the ratio between demands and supplies, which can be calculated as
\[
R = \frac{n \sum_{i=1}^{m} p_i c_i}{C}
\]

(1)

where $n$ is the current number of threads. When $R > 1$, the computing system will be overloaded and all the threads will be slowed down. We calculate the threshold of $n$ by
\[
n_{th} = \frac{C}{\sum_{i=1}^{m} p_i c_i}
\]

(2)

In our model, we assume when the system is congested, namely $n > n_{th}$, all the processors will be affected equally. Their running speed will be decreased by a factor of $R$. Therefore, the arrival and service rates of transactions ($\lambda$ and $\mu$) running on these processors will also be decreased by a factor of $R$:
\[
\lambda_{cor} = \begin{cases} 
\lambda, & n \leq n_{th} \\
\lambda/R, & \text{otherwise}
\end{cases}
\]

(3)

\[
\mu_{cor} = \begin{cases} 
\mu, & n \leq n_{th} \\
\mu/R, & \text{otherwise}
\end{cases}
\]

(4)

and we use these corrected $\lambda_{cor}$ and $\mu_{cor}$ in the following analysis.

C. Queuing Model of TM systems

With the above notational preparation and initial analysis on the execution of transactions, we can model the TM system as a finite source queuing system where each transaction is a client and the computing system acts as the server responding to the clients, and the client in our system may quit a current service and repeat the service process again (transactions may abort at a checkpoint and re-execute from the beginning). The states of the TM systems is shown in Fig. 3, where $N$ is the number of threads and $K = \max(k_1, k_2, ..., k_m)$ represents the maximum number of checkpoints that transactions may have in the target TM system. (Note that $k_i$ denotes the number of checkpoints for transaction $X_i$). The states are defined as follows:

- **State $[0]$** denotes the status of the system where no thread is executing any transactions.
- **State $[n,x]$** denotes the status of the system where (1) there exist $n$ transactions; (2) at least one transaction will commit; and (3) the first transaction that will commit is working towards its $x^{th}$ checkpoint. We denote the transaction that will commit first as the working transaction, and the thread executing it the working thread.
- **State $[n]$** denotes the status of the system where (1) there exist $n$ transactions; and (2) all will abort and restart at the next checkpoint (or commit point). According to the conflict resolution strategy of our targeted TM systems, the aborted transaction will restart immediately after the abort. The transaction that can commit earlier than others will become the new working transaction, which is denoted by quasi-working transaction, and the thread executing it is denoted by quasi-working thread. Note that the “quasi-working thread” is not necessarily the one that was first aborted, but it must be able to commit first.

Based on above definition of states, the possible transition routes among the states are presented in Fig. 3. Each of these routes is based on a possible system transition invoked by the arrival of an event. As defined in Section IV-A, there are three possible types of events: Start, Checkpoint and Commit. Note that we only analyze the events issued by the working thread, quasi-working thread or potential thread according to our state definition, because other events will not cause state transitions. Except for Start, each event has two possible consequences for the current system state, “Succeed” and “Abort”. “Succeed” means the transaction issuing this event will move on to the next checkpoint. “Abort” denotes the scenario that the arrival event causes the system to abandon the previous work, which means the transaction fails its validity check and will rollback to the start point immediately. Start event does not have the “Abort” consequence because, as we assumed in Section III, if a thread fails to start, it will re-start the transaction immediately until it succeeds.

In Fig. 3, different colors are used to distinguish the transitions caused by different types of events: yellow lines represent the transition caused by Start events, blue lines are for the Commit events when they “Succeed”, black lines are for Checkpoint events when they “Succeed”, and red lines are for both Commit and Checkpoint events when they “Abort”.

In particular, the transitions invoked by Start and successful Commit events have multiple cases. When a thread issues a new transaction at the state $[n,x]$, the number of threads in the system will be incremented by 1, from $n$ to $n + 1$. However, depending on whether the new thread can replace the current working thread or not (if the new transaction can commit early than the current one), the state will transit to either $[n,1]$ or $[n,x]$. For a Commit event that succeeds, we also have multiple destinations for the transition. When the working transaction commits, the number of threads in the system is decremented by 1. Depending on the status of the next working transaction, the destination varies: if there is no working transaction (all transactions will be aborted), the destination state is $[(n-1),y]$; if the next working transaction is approaching its $y^{th}$ checkpoint, the destination state is $[(n-1),y]$.

Furthermore, not all the states can see all three types of events. For instance, when the system is at $[0]$, the only possible coming event is a Start, because no transaction exists in the system at that moment. Similarly, states $[n,K]$ have no Checkpoint arrivals and states $[N,x]$ have no
Start events.

D. Derivation of Transition Rates

After identifying the possible transition routes, we next examine the transition rates for each route. We simply enumerate every possibility that may occur at each state, and calculate the rates along the same routes.

During the transactional execution, three types of events exist: Start, Commit and Checkpoint. The Commit and Checkpoint events can cause two consequences: “Succeed” or “Abort”. Every Start and “Succeed”ing Commit event may have multiple transition routes as shown in Fig. 3. The derivation of transition rates is based on the probability analysis of these cases.

1) Start Event: Start occurs when a potential thread issues a new transaction. All states except \([N, x]\) and \([N'](x \in \mathbb{Z}, 1 \leq x \leq K)\) may have a Start event. Two possible outcomes are listed below:

- Succeed:
  
  The Start event may lead system state transit to two different states. The total arrival rate of these two routes is calculated by counting how many threads are NOT executing transactions, because every one of them has the potential to issue one. For those states \([n, x]\) or \([n']\) having \([N - n]\) potential threads, the total arrival rate is

  \[
  \lambda_{(n)} = (N - n) \lambda_{cor} \tag{5}
  \]

  Next, we differentiate two cases of transitions when a new Start arrives on states \([n, x]\) and \([n']\) respectively. States \([n, x]\) will transit 1) right to \([n + 1, x]\) or 2) down to \([n + 1, 1]\). States \([n']\) will transit 1) right to \([(n + 1)']\) or 2) up to \([n + 1, 1]\). The key affecting the transition rates is whether the new transaction will take place of the current working transaction (for the states \([n, x]\) or commit early than the quasi-working thread (for the states \([n']\)).

For states \([n, x]\), let us say the new transaction \(X_n\) has \(k_n\) checkpoints with service rate \(\mu_n\), and the working thread is executing a transaction \(X_b\) with \(k_b\) checkpoints and service rate \(\mu_b\). The rate of the corresponding transition from \([n, x]\) to \([n + 1, 1]\) can be calculated by

\[
\lambda^{x-1}_{(n,x)} = \lambda_{(n)}(1 - p_{con})^n \sum_{a=1}^m p_a p_b P(X_a < X_b)dt \tag{6}
\]

where \(P(X_a < X_b)\) is the probability that transaction \(X_a\) will finish before transaction \(X_b\) (\(X_a\) will replace \(X_b\) to be the new working transaction). We can calculate it by deriving the probability of a Erlang-\((k_a, \mu_a)\) distributed variable smaller than another Erlang-\((k_b - x, \mu_b)\) distributed variable.

\[
P(X_a < X_b) = \int \frac{\Gamma(k_a, \mu_a t)}{(k_a - 1)!} \cdot \frac{\mu_b^{k_b} e^{-\mu_b t}}{(k_b - 1)!} dt \tag{7}
\]

On the other hand, the rate of transition from \([n, x]\) to \([n + 1, x]\) is

\[
\lambda_{(n,x)} = \lambda_{(n)} - \lambda^{x-1}_{(n,x)} \tag{8}
\]

States \([n']\) have similar cases and the calculation are identical to Eq. 6 and 8. We only need to set \(x = -1\) (for the extra work due to rollback) when using the equations.

- Abort: States will not be changed since if a thread fails to Start, then it will restart the transaction immediately and continuously retry until succeed. We treat these retries in a very short instant as a single event.

2) Commit Event: A Commit denotes an executing transaction reaching its final checkpoint. We only focus on the Commit events issued by a working thread (blue lines) or a quasi-working thread (red lines) because their Commit events must arrive before all other threads according to the
definition. Moreover, by definition, the only outcome of the Commit event issued by the working thread is “Succeed”. All states except state $[0]$ may have a Commit event and two outcomes are listed below:

- **Succeed**: Only states $[n, x]$ that $x \in k_1, k_2, \ldots, k_m$ may encounter a Commit Event. When a Commit Event arrives, there are multiple transition routes available (except for states $[1, x]$ whose two routes are merged). Same as the derivation for Start events, we calculate the total commit rate $\gamma(n)$ of each state $[n, x]$ by summing over the commit rates of all transactions whose $k_i = x$

$$\gamma(n) = ((n - 1)(1 - p_{con}) + 1) \mu_{cor} \sum_{k_i = x} p_i k_i \quad (9)$$

Next, we examine different routes of transitions. For state $[n, x]$, we differentiate out two basic scenarios by the destinations: $[n - 1, y]$ or $(n - 1)'y$. When the current working transaction commits, among the remaining $n - 1$ threads, there is a probability of $(1 - p_{con})$ that there exist a non-conflicting transaction that will commit afterwards, and become the new working transaction. The states therefore will transit to $[n - 1, y], 1 \leq y \leq K$ where $y$ denotes that the new working thread is working towards its $y^{th}$ checkpoint. Since this is a non-conflicting transaction, every checkpoint of it will be passed only once. Thus, at a given time, the probabilities that this transaction is working towards a certain checkpoint should be equal. We use $X_a$ to denote the committing transaction and $X_b$ denote the next working transaction. Based on above analysis, we derive the rate of transition from state $[n, x]$ to each $[n, y]$:

$$\gamma^{x-y}_{(n,x)} = \sum_{k_a = x} \sum_{y = 1}^m \frac{1}{k_b} (1 - p_{con}) p_a p_i \gamma(n) \quad (10)$$

which can be used to calculate the rates for the top $K$ commit routes for states $[n, x]$. The other case is that if no such non-conflicting transaction exists, all of the rest $n - 1$ transactions should be aborted sooner or later. As a result, the transition will be led to $(n - 1)'y$. The rate can be calculated by

$$\gamma^{n-(n-1)'}_{(n,x)} = \gamma(n) - \sum_{y=1}^i \gamma^{x-y}_{(n,x)} \quad (11)$$

- **Abort**: A Commit arrives (from the quasi-working thread) at states $[n']$ will lead to “Abort”. In fact, when the system is at states $[n']$, the Commit and Checkpoint events have the same effect: the validity check will let the transaction abort and restart. Thus, they share the same routes (red lines in Fig. 3) pointing to states $[n, 1]$. We combine the transition rates due to these two types of events and acquire as follows:

$$\gamma^{n'-(n,1)}_{(n,x)} = ((n - 1)(1 - p_{con}) + 1) \mu_{cor} \quad (12)$$

3) Checkpoint Event: A Checkpoint event means the running transaction is doing a validity check. We only focus on the Checkpoint events issued by a working thread (black lines) or a quasi-working thread (red lines) because other Checkpoint events will not cause state transitions. All states except states $[0]$ and $[n, K], (n \in N, 1 \leq n \leq N)$ may have a Checkpoint event and two consequences are listed below.

- **Succeed**: The outcome of the Checkpoint events issued from states $[n, x]$ must be “Succeed”. The only destination of the transition is states $[n, x + 1]$, and the rate is

$$\omega_{(n,x)} = ((n - 1)(1 - p_{con}) + 1) \mu_{cor} \sum_{k_i \geq x} p_i k_i \quad (13)$$

- **Abort**: A Checkpoint arrives (from the quasi-working thread) at states $[n']$ will lead a ‘Abort’. The discussion of this transition rate is combined in discussion of ‘Abort’ outcome of Commit event (Section IV-D2).

E. Transaction Execution Time

Let $\pi$ denote the steady state probability vector $[P(0), P(1,1), P(1,2), \ldots, P(2,1), \ldots]$ where $P(X)$ denotes the steady probability of state $(X)$. The intensity matrix $Q$ records the transition rate between the states, which can be calculated by following every transition route in Fig. 3 (omitted here due to space limit). For the steady state probability, we have

$$\left\{ \begin{array}{l}
\pi Q = 0 \\
\sum_{p \in \pi} p = 1
\end{array} \right. \quad (14)$$

After solving Eq. 14, we can derive the steady-state probabilities for all states. Note that although it is difficult to obtain a closed form solution for $\pi$ from Eq. 14, a numerical solution can be obtained easily. The expected number of threads in the system is therefore

$$E(L) = \sum_{X \in S} n_X P_X \quad (15)$$

where $S$ is the collection of all states in Fig. 3, $n_X$ is the number of transactions when system state is $X$, and $P_X$ is the probability of state $X$. Similarly, the mean arrival rate is $E(\lambda) = \sum_{X \in S} \lambda_X P_X$ where $\lambda_X$ is the arrival rate when system is at state $X$. By Little’s Law, we derive the expected transaction execution time and use it to represent the performance of the underlying TM system:


\[ E(W) = \frac{E(L)}{E(\lambda)} = \frac{\sum_{X \in S} n_X P_X}{\sum_{X \in S} \lambda_X P_X} \] (16)

which can be calculated numerically.

V. EXPERIMENTS AND DISCUSSIONS

We validate our model against the STAMP benchmark suite, which is widely accepted by the researchers as the typical TM workload. STAMP has an open infrastructure that allows the use of various TM implementations, and we tested TL2, TinySTM and SwissTM [13]. These are three state-of-the-art TM implementations that have demonstrated good performance. The STAMP benchmark programs are configured with the suggested configurations in [4]. Unless otherwise indicated, TL2 is configured with the ‘lazy’ option (featuring lazy conflict detection and resolution). TinySTM is configured with the ‘ETL-WB-Suicide’ option (featuring encounter-time-locking, write-back, and suicide upon detection of conflict), and SwissTM by itself is configured with a mixed conflict detection strategy (eager WAW and lazy RAW conflict detection, transactions having less work will be aborted) \(^1\). The experiment platform had 24 cores (4 Intel Six-core Xeon E7450 2.4GHz) and 64GB DDR2 800Hz shared memory. The operating system (Redhat Enterprise 5 distribution 64-bit) ran Linux kernel version 2.6.18 and gcc version 4.1.2 was used to generate the executables.

We first examine the accuracy of our model when modeling different applications. TinySTM is selected as the TM system. During the experiments, we first profile the STAMP benchmarks with 1 thread, then use our model to predict their performance of 2, 4, 8 and 16 threads. The prediction results are compared against the real execution results. STAMP contains a list of benchmarks with different characteristics as listed in [4]. The characteristics include the percentage of transactional execution time, transaction length, and average retries per transaction (contention level). The characteristics in [4] were obtained using simulations based on TL2, which deviate slightly from our results on real machines using TinySTM. In particular, we used hardware Time Stamp Counter (TSC) on x86 architectures as the high resolution timer. Although this is the lowest overhead timer that we are aware of, time measurement inevitably introduces certain instrumentation overheads for the TM system. The impact of such overhead is negligible for most benchmark programs as their transactions consist of thousands of instructions, much longer than the instrumentation. But for benchmark programs with very short transactions (kmeans and ssca2), the impact is noticeable. The transactions were prolonged by the instrumentation. In our analysis, we treat such instrumentation overheads as part of the overheads associated with the TM implementations.

\(^1\) Although TL2 and SwissTM adopt different conflict resolution strategies other than “suicide”, the prediction error of our model still relatively low. So we include their results as well.

The parameters describing the properties of the benchmark programs are summarized in Table I and were used as inputs to our model. The number of transaction types \(m\) was obtained by examining the source code. We also measured \(p_i\) for each type of transactions by calculating the number of tries they had to make before the commit. We fixed the service rate \(\mu = 1\) in our calculations as we use the normalized transaction execution time as the performance metric. According to the definition in Section III, we can calculate the actual \(\lambda\) as follows:

\[ \lambda = \frac{T \mu}{1 - T} \] (17)

where \(T\) is the time spent in transactions for each benchmark (Time in Tx in Table I). The contention level reflects the conflicting probability \(p_{\text{con}}\), and it is related to the average retries per transaction that can be observed. The ‘consumption’ row in Table I depicts the demands on the resources. It is estimated by checking the ratio of the number of read/write barriers over the transaction length, all of which can be found in [4]. When we set the total computing resources \(C\) to 24 (assuming the computer can support 24 cores without performance degradation when each of them consumes a maximal of \(c_i = 1\) unit of resources), we choose the values of \(c_i\) according to its consumptions and the average number of \(c_i\) is presented in Table I, where a value larger than 1 represents a higher computing resources consumption than normal code.

The results of actual execution and our model based prediction are illustrated in Fig. 4. We normalize the mean transaction execution time to that of a single thread. It can be seen that our theoretical prediction fits actual executions very well. The average error rate is 7.9%. The maximum error rate is 14.7% for \(k\text{means}-\text{low}\) with 16 threads. In both Fig. 4, \(k\text{means}-\text{high}\) and \(\text{genome}\) have the worst and best scalability respectively, which are mainly determined by their contention levels. Similar results can be found in [22]. Note that we present the mean transaction execution time, while other profiling results present the total execution time which also includes the time costs of the sequential portions.

Next we verify the capability of our model in modeling different TM systems. Although different conflict resolution strategies are adopted in TL2 and SwissTM, our model still exhibits good prediction accuracy. We focus on the intruder benchmark because it has three types of transactions with equal percentages and the length of transactions are roughly the same. This gives us great opportunity to easily tune the number of checkpoints \(k_i\) in our model. We use the same value for \(k_1, k_2\) to \(k_3\) and use \(k\) for short in the following discussion. We tested all three TM systems: TL2, TinySTM and SwissTM. Because TL2 is configured with lazy conflict detection and resolution, it will even tolerate WAW conflicts so that it has the smallest number of checkpoints and least overheads associated with checkpointing. TinySTM is able
Table I

MODEL INPUT PARAMETERS

<table>
<thead>
<tr>
<th></th>
<th>genome</th>
<th>intruder</th>
<th>kmeans-high</th>
<th>kmeans-low</th>
<th>labyrinth</th>
<th>scea2</th>
<th>vacation-high</th>
<th>vacation-low</th>
<th>yada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in Tx</td>
<td>97%</td>
<td>43%</td>
<td>33%</td>
<td>30%</td>
<td>96%</td>
<td>25%</td>
<td>86%</td>
<td>86%</td>
<td>97%</td>
</tr>
<tr>
<td>$\lambda/\mu$</td>
<td>32.3</td>
<td>0.75</td>
<td>0.5</td>
<td>0.4</td>
<td>24</td>
<td>0.33</td>
<td>6.1</td>
<td>6.1</td>
<td>32.3</td>
</tr>
<tr>
<td>Contention</td>
<td>Low</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Low</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>$p_{\text{con}}$</td>
<td>0.10</td>
<td>0.50</td>
<td>0.90</td>
<td>0.75</td>
<td>0.52</td>
<td>0.43</td>
<td>0.26</td>
<td>0.25</td>
<td>0.70</td>
</tr>
<tr>
<td>Consumption</td>
<td>Low</td>
<td>High</td>
<td>Normal</td>
<td>Normal</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>$\bar{c}_i$</td>
<td>0.9</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2.1</td>
<td>1.2</td>
<td>1.2</td>
<td>2.2</td>
</tr>
<tr>
<td>Tx Types ($m$)</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Percentage of Tx</td>
<td>0.01, 0.50, 0.34, 0.33, 0.41, 0.07, 0.33</td>
<td>0.75, 0.01, 0.24</td>
<td>0.75, 0.01, 0.24</td>
<td>0.5, 0.001, 0.499, 0.005</td>
<td>0.005, 0.99, 0.005</td>
<td>0.98, 0.01, 0.05, 0.01</td>
<td>0.90, 0.05, 0.01, 0.17, 0.24, 0.17</td>
<td>0.5, 0.001, 0.499, 0.005</td>
<td>0.98, 0.01, 0.05, 0.01, 0.17, 0.24, 0.17</td>
</tr>
</tbody>
</table>

* bayes benchmark is excluded because of its non-deterministic finishing conditions as noted in [4], which makes the comparison against the deterministic result generated by theoretical model less meaningful.

Figure 4. Comparison between real executions and theoretical prediction for different benchmarks

Figure 5. Comparison between real executions and theoretical prediction for different TM implementations

to detect most conflicts, thus it has the largest number of checkpoints and also with the highest overheads. The number of checkpoints and overheads of SwissTM is between TL2 and TinySTM because it has a mixed conflict detection strategy. Therefore, we set $k$ for TL2 to 1, TinySTM to 15, and SwissTM to 10, and $\bar{c}_i$ for TL2 to 1, TinySTM to 4, and SwissTM to 3. Fig. 5 compares actual execution results against our model prediction. The results show that TL2 has an exponentially degrading performance when the number of threads increases. We believe this is primarily due to the lazy option used in TL2 that causes a transaction to waste time in computation that will eventually be aborted. TinySTM and SwissTM have comparable performance on intruder, but because TinySTM invokes more conflict detection than SwissTM (larger number of checkpoints), SwissTM is a little bit faster than TinySTM especially when the number of threads increases. Our model can describe these differences and make the prediction very close to the real performance.

VI. CONCLUSION

In this paper, we developed a novel theoretical model to predict the performance of TM systems. Based on the statistical characteristics of transactions, CTMC is employed to model the TM systems: every transaction is a client requesting services and the computer is the server responding these requests. The start, commit and checkpoint events in transactional execution are directly mapped in our model to represent state transitions in the TM systems. By calculating the probability of every state where the TM system may stay, we obtain the mean transaction execution time to evaluate the performance of target TM system. Our model is validated through extensive experiments comparing the results of real execution against our theoretical prediction.

Our model achieved an average error rate of 7.9% in the comparison against real TM systems for STAMP benchmarks. Different from previous performance models, our CTMC model can capture the run-time behaviors of TM systems. However, our model still needs work in following directions: 1) simplifying the input parameters of model because acquiring some of them is a non-trivial task, and 2) verifying our model with other TM systems including HTMs.
ACKNOWLEDGMENT

This work is supported by the US National Science Foundation under award number CNS-0845583. We also thank Jean Daniel Mathieu and Xiao Yu for their contributions on the experiments.

REFERENCES


