Impact of Early Abort Mechanisms on Lock-Based Software Transactional Memory

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

Abstract

Software transactional memory (STM) is an emerging concurrency control mechanism for shared memory accesses. Early abort is one of the important techniques to improve the execution speed of STMs and has been explored intensively via experimental studies. This paper presents a theoretical analysis characterizing the properties of early abort and its impact on the performance of lock-based STMs. Queuing theory is adopted to model the behaviors of transactional execution. Analytical results are obtained for STMs with and without early abort. The analysis is validated through extensive experiments. Our results reveal that although early abort helps improve the performance of lock-based STMs especially when the contention level is low, the gain is often marginal. We expect our theoretical results to provide useful guidance towards the design and selection of appropriate lock-based STM schemes.

1. Introduction

Recently, transactional memory (TM) has emerged as a promising new paradigm for parallel programming. It offers much improved programmability over the traditional concurrency control mechanism of locks. It is expected to significantly reduce the difficulties of parallel programming and debugging, the vulnerability to failures and faults, and the likelihood of deadlocks. With the rapid trend shifting towards multi-core or multi-processor computing systems, more and more research efforts are being dedicated to the investigation of TM, especially the software transactional memory (STM).

Despite its advantages in programmability, in practice STM suffers a performance hit by as much as 50% relative to fine-grained lock-based code [6]. The primary reason for the low performance is due to the extra overheads associated with maintaining the modification logs and aborting/committing the transactions. Consequently, various designs have been proposed to improve the execution speed of STMs: STM was first developed with lock-freedom [10], then gradually evolves to obstruction-freedom [12, 14] and now lock-based [6, 8, 9, 19]. Currently, lock-based STM is considered to be faster than other STM designs. Many types of trade-offs are being explored in the design of lock-based STMs, which include transaction granularity (word- or object-based), update strategy (direct or deferred), conflict detection (eager or lazy), conflict resolution (abort or helping), etc [17] (Other terms may be used to describe these characteristics and we adopt the terms in [17]).

Object-based STMs map data to metadata at the granularity of objects which might have different sizes. The advantage of this arrangement is the data locality and the option of only protecting the data that is actually used. Word-based STMs protect the data on a per-word basis. With this arrangement, STMs do not need to track the individual code segments that have accessed the protected data. In this paper we focus on the word-based schemes. Note that our results are also applicable to object-based STMs due to their support of early abort mechanisms.

Deferred updates are often adopted by the commit-time-locking (CTL) designs in which large write/read sets are maintained until the transaction commits [6]. On the contrary, direct updates need to lock the data for every access (for versioning), which is also known as encounter-time-locking (ETL) \(^1\). These two updating strategies lead to two conflict detection strategies. ETL is often associated with eager conflict detection, because the encounter time lock acquisitions can make the accesses readily visible to the other transactions. CTL, on the other hand, is often associated with lazy conflict detection. When conflicts are detected, plenty of resolution methods have been proposed which include abort, helping or even global commit coordination. Abort is the most widely used method due to its efficiency and simplicity in implementation.

Among the above design options, the strategies of conflict detections and resolutions have attracted intensive research interests. Early abort (eager conflict detection and aborting a transaction to resolve the conflict) would save the computa-

---

1. There are some cases that ETL features deferred updates (e.g. in [9] where it was denoted as write-back).
tional resources which might be wasted if an uncommitable transaction continues to execute. But some studies argue that the overheads of supporting early abort will counteract this benefit. In addition, lazy conflict detection may help avoid some read-write conflicts. The benefit of early abort is therefore still inconclusive.

The objective of this paper is to quantify the impact of early abort through theoretical analysis. This approach differs from most existing studies where the focus is on the design and implementation of STMs, and the performance impact of early abort (as well as other STM design options) is evaluated through experimental methods via benchmark suites such as STAMP [5] or StmBench7 [11]. Such empirical evaluation methods do provide very useful insight to STMs, but it is often affected by the overheads associated with the programming implementations (see Section 4 for a detailed example where different implementations exhibit inconsistencies in execution efficiency). To the best of our knowledge, we are not aware of any prior work on the theoretical analysis of STMs. We expect our results to provide useful insights to understanding the impact of early abort.

We formally characterize early abort as: if a STM design allows a transaction to be aborted before it reaches the commit point (where the changes made by a transaction are to be validated and made permanent), the design belongs to the early abort category and is denoted as STM-EA; if a STM design only allows a transaction to be aborted at the commit point, the design belongs to the none early abort category and is denoted as STM-NEA.

Our goal is to investigate the impact of early abort from a theoretical point of view. We establish statistical models for STM-EA and STM-NEA. Queuing models are employed to describe transactional executions in STM-EA and STM-NEA. Our model considers the number of threads, the rate at which transaction are issued, the frequency of conflict detections, and the bandwidth of the interconnect. Transaction contention is modeled through the feedback on the service rate at which the transactions are executed. The analytical result is validated against extensive experiments using the M5 multi-processor simulator [4].

Because our focus is on the impact of early abort, we omit the following implementation overheads in our analysis: check pointing to decide whether a transaction should advance or abort, maintaining the undo list, and aborting. Our analysis effectively models the asymptotic behavior of STM-EA and STM-NEA where such implementation overheads have already been minimized. Our analysis therefore complements the existing studies (which focus on the design and implementations STMs) from the theoretical aspect. The inclusion of these overheads will positively improve the model and we have planned to do so in our future studies.

Our study shows that early abort allows STM-EA to slightly outperform STM-NEA when the contention level is low, and that the performance gain is marginal. Our results indicate that STM-EA and STM-NEA are suitable for different situations: when the system contention level is low and performance is the most significant consideration, STM-EA should be preferred; for other situations, STM-NEA would be a good choice due to reduced implementation complexity. We expect our theoretical results to provide useful guidance towards the selection of appropriate STM schemes, especially in the early stages of designing parallel applications.

The rest of the paper is organized as follows: Section 2 summarizes background and related work. Section 3 introduces our modeling and analysis of STM. Section 4 presents the numerical analysis and the simulation based experimental results. Section 5 is the conclusion and discussions.

2. Background and Related Work

The concept of transactions 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 [16] and has since been explored [1, 3]. Software-only transactional memory has recently been the focus of intense research, and support for practical implementations is growing [13, 15, 20]. Schemes that mix hardware and software have also been explored [7, 18, 21].

In STM, a transaction completes modifications to shared memory regardless of what other threads might be doing. Reads and writes inside transactions should logically occur at a single instant in time. No intermediate states can be observed or interfered by peers. Every thread records its read and write operations in a log. Instead of requiring the writer to make sure it does not adversely affect other operations in progress, readers take the responsibility: after completing an entire transaction, readers need to verify that other threads have not concurrently made changes to memory that was accessed in the past. This final operation, in which the changes of a transaction are validated and made permanent, if validation is successful, is called a commit.

STM is typically utilized in the following form:

```c
while (true) {
    STM_begin()
    try {
        critical section()
    } catch STM_invalid { continue }
    STM_commit()
    break
}
```

Improving the performance of STM has been the focus of intensive research. STM designs have evolved from lock-free to obstruction-free, and to lock-based. Ennals [8] observed
that non-blocking transactions are unnecessary for practical uses and demonstrated the advantages of lock-based STM through experiments. Ennals' STM is an encounter-time-locking (ETL) based STM. Saha et al. [19] implemented their McRT-STM in a runtime system. Similar to [8], the scheme in [19] requests locks at encounter time and uses pessimistic reads and writes. A transaction always attempts to acquire exclusive ownership (via locks) before it reads a variable, failure to do so will cause it to abort and roll back. Write operations in [19] are handled similarly, an undo log is maintained in case the transaction needs to rollback. Both schemes feature the early abort strategy - meaning that a transaction may abort any time before reaching the commit point.

Dice et al. [6] implemented Transactional Locking II (TL2), a commit-time locking (CTL) based STM scheme. TL2 then evolves to TL2C [2] with the adoption of a distributed clock. In both TL2 and TL2C, locks are acquired only during commit time. Read and write operations in a transaction are recorded in a read/write-set. During commit time, all read/write locations are locked, and version numbers of the memory locations are checked to determine whether the transaction should commit or abort. TL2C uses deferred update (write-back) strategy, buffering all the updates locally at each thread before permanently writing them to the shared memory locations. TL2C is considered to be one of the faster STM designs.

Recently, ETL is revisited by Felber[9] et al. A deferred update strategy is introduced where the memory location is locked at encounter time but the updates are stored in a write-set and validated only during commit time. This is a hybrid ETL/CTL scheme that employs eager conflict detection to lower the cost of aborting. Nevertheless, experiments demonstrated similar performance between the direct update (write-through) and deferred update (write-back) schemes.

3. Analysis of STMs with Early Abort

In this section, we first establish a model for STM-EA and then extend it to STM-NEA. We derive mean transaction completion time for the two schemes.

3.1. System Models

We focus on an abstract scenario where all the threads execute the same transactions, i.e. all transactions perform the same operations on the same read and write sets. Interleaved or nested transactions, or transactions with semi-overlapped read/write sets have complicated behaviors and we leave them for our future studies.

Fig. 1 illustrates the time diagram when executing the transactions. Without loss of generality, we assumed that threads T1, T2, T3 and T4 enter their transactions in the illustrated order. T2, T3 and T4 enter their transactions after T1 starts, and therefore cannot commit. After T1 commits its transaction, the other threads can detect the conflict after certain amount of time, the length of which depends on the specific conflict detection scheme. STM-EA detects the conflicts earlier than STM-NEA because STM-EA checks more frequently and is able to detect the conflicts right after the failures of lock acquiring or version number checking. On the contrary, STM-NEA will let T2, T3, and T4 continue working on their current transactions until they need to commit. For example, when T2's transaction is about to commit, T2 will find that T1 has modified the variables it recorded, and therefore will rollback and re-execute its transaction. Next, T2 can commit the transaction successfully but T3 has to rollback once more and so does T4. Due to their strategies in detecting conflicts, STM-EA and STM-NEA cause the system to waste different amounts of time, during which all the threads perform useless computations (that will abort). The wasted time further affects the number of active threads competing for resources in the system, and will hence affect the rate at which the transactions are executed. In the next section, we quantify the amounts and their impacts on the overall performance.

At a given time instance, we may or may not have any transactional events (starting, check-pointing, committing, and aborting, etc.), which can be abstracted using a binomial distribution. When the number of events becomes big and the probability of occurrence stays low, the binomial distribution approaches a Poisson distribution and the intervals between two Poisson events are exponentially distributed. Similar to networking models, exponential distribution is an excellent choice to describe such behaviors from the statistical aspect. With this observation, we consider the following scenario in our analysis:

1) The system consists of N processor cores sharing the main memory, where each core executes a single thread. Every thread repeatedly issues transactions with a Poisson distribution, i.e., for each thread, the time period between two consecutive transactions obeys an exponential distribution.
2) All the transactions access the same memory locations. Therefore, when multiple threads execute their transactions simultaneously, the thread that finishes the computation first will be able to commit and all others will have to rollback.

3) For each transaction, the time required to complete it (denote as the transaction service rate) is also a random variable with exponential distribution. The service rate for any particular transaction depends on the number of active threads that are executing the transactions. This reflects the impact of multiple threads competing for the interconnect accommodating all memory accesses.

4) The reads and writes are uniformly located in the transaction, and the intervals between the check points obey exponential distribution. This means that transaction abort also obeys a Poisson distribution, at a rate higher than the transaction service rate. Granularity of the check points determines the transaction abort rate. We use $T$ to denote the ratio between transaction abort rate to transaction service rate.

With the above assumptions, STM-NEA can be considered as a special case of STM-EA where $T$ is set to 1. $T = 1$ means transactions only have one chance (at commit time) to check whether they need to abort, which is essentially an STM without early abort.

### 3.2. Queuing Analysis

Based on the above assumptions, we modeled the system as a finite source queuing system where the interconnect acts as the servers answering requests from the processors. We abstract away the topology of the interconnect by characterizing it using its bandwidth $B$. We employ Continuous Time Markov Chain (CTMC) to build the state transition diagram as shown in Fig. 2.

To model the abort and rollback behavior, we differentiate two scenarios. We say the system is at state $i'$ if there exist $i$ threads, and exactly one of them (denoted as the working thread) will commit its transaction and the others (denoted as the interfering threads) will rollback. We say the system is at state $i''$ if there exist $i$ threads and all of them will rollback their transaction at either a check point or the commit point. The steady-state probability that the system is at state $i(i'')$ is denoted as $P_i(P''_i)$.

Next we obtain the transfer rates between the states.

#### 3.2.1. Arrival Rate

At every state, a thread that is not executing a transaction may issue one. The net arrival rate (after combining all such threads) is determined by the total number of threads that are not executing any transactions. Therefore, when multiple threads execute their transactions, we can safely assume that these transactions were launched at $t_0$, which is the time instance that the new transaction (e.g. $T_1$ in Fig. 3) issued their current transactions, we can safely assume that these transactions were launched at $t_0$, which is the time instance that the new transaction (e.g. $T_1$ in Fig. 3)
arrives. All these transactions will abort (since this is state $i'$) at the transaction abort rate (e.g. $t_0 \rightarrow t_{21}$ for $T2$ in Fig. 3), and re-execute at transaction service rate (e.g. $t_{21} \rightarrow t_{22}$ for $T2$ in Fig. 3). The newly arrived transaction will proceed at the transaction service rate (e.g. $t_0 \rightarrow t_{11}$ for $T2$ in Fig. 3). For the new transaction to commit successfully, it needs to complete its transaction before all the existing transactions can complete their roll-back (as illustrated in 3(a)), otherwise, one of the existing transactions will commit (as illustrated in 3(b)). By lengthy probability derivation (omitted here due to space limitations), we obtain

$$Pr\{(i-1)' \rightarrow i\} = \frac{1}{i} \sum_{m=0}^{i-1} \prod_{l=1}^{i-1} \frac{(i-m)}{(i-m+mT)}$$

(3)

and

$$Pr\{(i-1)' \rightarrow i'\} = 1 - \frac{1}{i} \sum_{m=0}^{i-1} \prod_{l=1}^{i-1} \frac{(i-m)}{(i-m+mT)}$$

(4)

Subsequently, we obtain $\omega_i$ and $\omega_i'$ below:

$$\omega_i = \left(1 - \frac{1}{i+1} \sum_{l=1}^{i+1} \prod_{m=0}^{i-1} \frac{(i-m+1)}{(i-m+1+mT)}\right) \lambda_i$$

$$\omega_i' = \left(\frac{1}{i+1} \sum_{l=1}^{i+1} \prod_{m=0}^{i-1} \frac{(i-m+1)}{(i-m+1+mT)}\right) \lambda_i$$

(5)

3.2.2. Service Rate. In queuing theory terms, service rate determines the rate at which customers leave the system. Similarly in our transactional system, the service rate determines when a transaction is completed.

The overall serving capability of the system is affected by the processing power of the CPUs and the bandwidth of the interconnect (which connects the CPUs and the memory together). Recent advancement in microprocessor technology has created the well known memory wall where the processor speed is much faster than the memory and the interconnect. Consequently, we assume the serving capability is determined by the available bandwidth of the interconnect, which is evenly shared by all the active transactions. Let $B$ denote the bandwidth, the overall serving capability $\mu_i$ at state $i$ and $i'$ (both having $i$ active transactions) can be derived as:

$$\mu_i = \begin{cases} i\mu_0 & \text{if } i < B/\mu_0 \\ B & \text{otherwise} \end{cases} (i = 1, 2, 3, ..., N)$$

(6)

where $\mu_0$ denotes the bandwidth requirement of a single transaction.

Due to the fact that a transactions may be rolling back, the transaction may remain in the system even though it has consumed certain bandwidth. In other words, the overall serving capability is different from the actual serving rate, and system may stay at the same state even though some servicing event occurs in this queuing system. Consequently, different service rates need to be considered for the following four state transitions:

1) $i \rightarrow i$: When a interfering thread finishes its transaction and needs to rollback, the system will remain at state $i$. Its service rate is denoted by $\mu_{i-1}^{\prime\prime}$.

2) $i \rightarrow (i-1)'$: When a working thread finishes its transaction and the transaction commits successfully, the system will transit from state $i$ to $(i-1)'$ because the working thread has committed and all the other interfering threads will need to rollback. Its service rate is denoted as $\mu_{i-1}^{\prime\prime}$.

3) $i' \rightarrow i'$: When an interfering thread rolls back and starts its re-execution, another interfering thread may interfere it and let it abort. As a result, the system will remain in state $i'$. The service rate is denoted as $\gamma_i$.

4) $i' \rightarrow i$: When an interfering thread rolls back and if the re-execution will commit successfully, the system will transit from state $i'$ to $i$. The service rate is denoted as $\gamma_i'$.

We have:

$$\mu_i' + \mu_{i'}' = \mu_i \text{for } i = 1, 2, 3, ..., N$$

(7)

and because of the early abort feature of STM-EA, the service rate of state $i'$ is equal to the transaction abort rate $T\mu_i$:

$$\gamma_i + \gamma_{i'} = T\mu_i \text{for } i = 2, 3, ..., N$$

(8)
At state \( i \), the probability of case 2 is \( \frac{1}{i} \), and the probability of case 1 is \( \frac{i - 1}{i} \). Then

\[
\begin{align*}
\mu'_i &= \frac{i - 1}{i} \mu_i \\
\mu''_i &= \frac{1}{i} \mu_i
\end{align*}
\tag{9}
\]

Transitions 3 and 4 are very similar to the cases illustrated in Fig. 3, and their probabilities can be obtained similarly (omitted here due to page limit):

\[
Pr\{i' \rightarrow i\} = \frac{1}{i} \sum_{l=1}^{i-1} \prod_{m=0}^{l-1} \frac{(i - m)}{(i - m + mT)}
\tag{10}
\]

\[
Pr\{i' \rightarrow i\} = 1 - \frac{1}{i} \sum_{l=1}^{i-1} \prod_{m=0}^{l-1} \frac{(i - m)}{(i - m + mT)}
\tag{11}
\]

therefore,

\[
\begin{align*}
\gamma_i &= \left(1 - \frac{1}{i} \sum_{l=1}^{i} \prod_{m=0}^{l-1} \frac{(i - m)}{(i - m + mT)}\right) \mu_i \\
\gamma'_i &= \left(\frac{1}{i} \sum_{l=1}^{i} \prod_{m=0}^{l-1} \frac{(i - m)}{(i - m + mT)}\right) \mu_i
\end{align*}
\tag{12}
\]

3.2.3. Steady State Transaction Completion Time.

Let \( \Pi \) denote the steady state probability vector \([P_0, P_1, P'_1, P_2, ..., P_{N-1}, P'_{N-1}, P_N, P'_N]\). The intensity matrix \( Q \) (Eq. 13, on page 7) can be obtained by following the state diagram in Fig. 2.

For the steady state probability, we also have

\[
\begin{align*}
\Pi Q &= 0 \\
\sum_{i=0}^{N} P_i + \sum_{j=1}^{N} P'_j &= 1
\end{align*}
\tag{14}
\]

where \( I \) is the identity matrix and \( P_i(P'_j) \) is the probability that the system is at state \( i(i') \) in steady state. Substituting the results from Eq. 1, 5, 9 and 12 into intensity matrix and 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 Eq. 14, a numerical solution can be obtained easily. The expected number of threads in the system is therefore

\[
L = \sum_{i=0}^{N} i P_i + \sum_{j=1}^{N} j P'_j
\tag{15}
\]

and because the mean arrival rate is \( E(\lambda) = \sum_{i=0}^{N} \lambda_i P_i + \sum_{j=1}^{N} \lambda_j P'_j \), by Little’s Law, we derive the mean transaction completion time:

\[
W = \frac{L}{E(\lambda)} = \frac{\sum_{i=0}^{N} i P_i + \sum_{j=1}^{N} j P'_j}{\sum_{i=0}^{N} \lambda_i P_i + \sum_{j=1}^{N} \lambda_j P'_j}
\tag{16}
\]

which can be calculated numerically.

4. Numerical Analysis and Simulation Results

In this section, we present the numerical analysis based on the above models along with the experimental results generated by the M5 multi-processor simulator [4].

In the experiments, all the transactions are designed to execute the same code that operates on an array of 10k 32-bit words. The operations merely contain pair-wise swap of the array elements which are designed to stress the interconnect. Each thread executes multiple (200) transactions separated by random intervals and the reported mean transaction completion time is averaged over all the threads. The intervals between consecutive transactions obey exponential distribution.

We first implemented the above transactions using TinySTM [9] and TL2 [6] on a system with four 6-Core Intel Xeon 2.4GHz processors. Linux kernel version 2.6.18 and GCC version 4.1.1 were used. TinySTM can be configured as ETL with write-back (WT) or write-through (WB), both featuring early abort. TinySTM also provides a CTL-based implementation that does not support early abort. We experimented all three implementations and because TinySTM(WB-ETL) and Tiny(WB-ETL) have very close performance as predicted in [9], we only show the results for TinySTM(WB-ETL) and TinySTM(CTL) in Fig. 4, which indicates that TinySTM(WB-ETL) outperforms TinySTM(CTL) from 2 to 16 threads. TinySTM(WB-ETL) is a typical STM-EA and TinySTM(CTL) is STM-NEA.

However, with TL2, a different implementation than TinySTM, we observe different performance curves. When TL2 is configured as lazy conflict detection, we consider it does not support early abort and represents STM-NEA. The performance comparison between TinySTM(WB-ETL) and TL2(Lazy) is shown in Fig. 5. They have very close performance from 2 to 16 threads. We can see the inconsistency between Fig. 4 and Fig. 5 for the comparison of STM-EA and STM-NEA. We conjecture it is due to the different amount of overheads in the implementations of TinySTM and TL2. As practical implementations of STM-EA and STM-NEA vary significantly in the pattern and amount of overheads, experimental results on real machines cannot provide enough insights on the impact of early abort. In order to focus on early-abort and exclude the impact of other factors, we conducted extensive simulation-based experiments as follows.

For the simulation-based study, we implemented our STM similar to TinySTM with write-back, which has demonstrated competitive performance. Because we know in advance the operations involved in each transaction, we do not need to dynamically identify the set of read/write variables. We therefore statically code the locking and abort functionalities into the transactions. Because the read and write sets are static in our experiments, the overheads associated
\[
Q = \begin{bmatrix}
-\lambda_0 & \lambda_0 & 0 & 0 & 0 & 0 \\
\mu_0 & -\mu_1 - \mu_0 & 0 & 0 & 0 & 0 \\
0 & \mu_0 & -\mu_0 - \omega_1 & \lambda_1 & 0 & 0 & 0 \\
0 & 0 & \mu_1' - \lambda_3 & \omega_1 & \lambda_3 & 0 & 0 \\
0 & 0 & 0 & -\gamma_2' - \omega_2 & \omega_2 & \omega_2 & \ldots \\
. & . & . & . & . & . & \ldots 
\end{bmatrix}
\]
an individual processor starts new transactions and issues them onto the interconnect. It can be adjusted by changing the length of the intervals between transactions. $B$ is related to the number of transactions the interconnect can support under zero interference. $T$ is adjusted in our transaction code by controlling how often a transaction performs conflict detection.

We first tested scenarios with low contention level. We set $B = 3.2$Gbps, $\lambda_0 = 0.1$ and $T = 10$. Fig. 6 compares STM-EA and STM-NEA where the number of threads ranges from 2 to 16. Both numerical analysis and experimental results indicate that STM-EA outperforms STM-NEA under this low contention level (11% in simulations and 21% in numerical analysis). The observed performance gap is smaller in the experimental results than in the numerical analysis. This is due to the check pointing and logging overheads introduced by the transactions in the simulations.

We then increased the number of threads to 32 and 64 and the result is shown in Fig. 7. This scenario has relatively high contention levels due to the larger number of threads. The performance predicted by analytical results is within 26% of the simulation results. Despite the overheads we neglected in the analysis, our model predicts that the performance gap between STM-EA and STM-NEA is small (18%), as validated by the simulations (where the gap is 10%).

The contention level is also affected by the interconnect bandwidth. This is studied in the next set of experiments. We tested bandwidth of 160Mbps and 640Mbps. The results is shown in Fig. 8. For both bandwidth settings, the absolute transaction completion time increased significantly as the number of threads increases. Further more, both the analytical and the simulation results indicate that STM-EA is slightly better (16% analytically and 6% in simulations) than STM-NEA.

The impact of transaction issue rate is illustrated in Fig. 9. The experiments were conducted with 8 threads and 160Mbps bandwidth. The simulation results have higher transaction completion time because of the overheads involved. For this set of experiments, analytical results showed that STM-EA is about 33% better than STM-NEA, but simulation results showed less than 12% performance gain.
In this paper, we developed a theoretical model to describe the impact of early abort on the performance of lock-based STMs. The analytical results are validated through extensive simulation-based experiments. Our results show that while STMs with early abort do outperform their counterparts without early abort, the performance gain is marginal. Early abort performs well when the contention level is low. Given the much higher complexity of implementing early abort, our results suggest that early abort should be applied only when the system contention level is low and performance is the most significant concern - under such situations, the (high) implementation cost may be justifiable. Our results also suggest that the present debates between ETL and CTL, and between optimistic and pessimistic reads/writes should focus on the reduction of overheads, rather than on early abort itself. We expect our theoretical results to provide useful...
guidance towards the design and selection of appropriate lock-based STM schemes.

Note our study did not model the overheads of locking, logging, rolling back, and committing, etc. As part of our future research, we will embedding these overheads into our model to obtain a closer to reality analysis. When such overheads are incorporated into the model, we expect to observe further less performance gains with early abort.

References


