Abstract
To address the need for improved efficiency and to support an ever-increasing feature set, the specification and implementation of state-of-the-art transactional memory (TM) systems are vastly different than the pioneering systems from a decade ago. An unfortunate side-effect of this shift is that some of the early theoretical work on TM correctness does not apply to these new systems.

This paper presents a work in progress to show a full invalidation automaton only accepts schedules that are conflict serializable, a principle correctness criterion for TM. In this paper we present the formalization of a full invalidation automaton and the general direction of our approach. We are interested in proving full invalidation correct because it is a recently devised method for performing transactional opacity and conflict detection and has been shown to be efficient for InvalSTM, a software-only TM (STM) that is competitive with other state-of-the-art STMs.

1. Introduction

Full invalidation is a conflict detection strategy where a committing transaction resolves its conflicts with active transactions before it commits [1]. This differs from validation in that validation checks a committing transaction for conflicts with transactions that have already committed. Because of this, a validating TM must abort the committing transaction if conflicts are found between it and other transactions. With full invalidation, if a conflict is found between a committing transaction and other active transactions, the TM can stall or abort the committing transaction or it can abort the conflicting, active transactions.

This paper formally defines full invalidation, originally proposed by Gottschlich et. al [1], in terms of an automaton and it provides the early definitions and conjectures that we plan to use to prove full invalidation is conflict serializable. This paper is an early report and a work in progress on our general approach. It uses an adapted model originating from Lynch et. al’s I/O automata [5]. We use conflict serializability as our criterion of correctness as defined by Papadimitriou [7]. We also extend Papadimitriou’s definition of a conflict graph to include a new type of graph specifically for deferred update systems, which can produce higher transaction throughputs by removing false conflicts. Some of the definitions we use are more completely defined in the above work [5, 7] and in Ramadan et. al’s DSTM [8].

2. Preliminary Definitions

A history, or schedule [7], is a sequence of instantaneous events. For our automaton, events are read, write, commit, or abort, as defined below:

- \( T \text{ commit, } I, W \): \( T \) commits with the set of \( I \) invalidated transactions and a set \( W \) of ordered pairs \( (x, v) \) where \( x \) is a variable and \( v \) is \( x \)’s associated written value.
- \( T \text{ abort} \): \( T \) aborts.

An event \( e_1 \) is said to happen before \( e_2 \) in history \( h \), denoted by \( e_1 <_h e_2 \), if \( e_1 \) occurs in \( h \) before \( e_2 \) occurs in \( h \). A transaction \( T_1 \) is said to happen before \( T_2 \) in history \( h \), denoted by \( T_1 <_h T_2 \), if \( T_1 \)'s commit event in \( h \) occurs before \( T_2 \)'s commit event in \( h \). A history is serializable if the total order of committed transactions is identical to some serial history of the transactions, where each transaction is executed one after another without interleaving any events from distinct transactions.

Informally, a conflict between two transactions exist when one active transaction, \( T_w \), writes to a memory element and another active transaction, \( T_{rw} \), reads or writes the same memory element and both transactions complete. Two histories are conflict equivalent if the conflicting events in the two histories have the same happens order.

A transaction \( T \) is conflict equivalent to a serial history if \( h \) is a history, \( T_1 \) and \( T_2 \) are distinct transactions [4], and \( x \) is a variable, then:

- \( \pi_1(P) \) returns the set of the first coordinates of the ordered pair \( P \).
- \( \pi_2(P) \) returns the set of the second coordinates of the ordered pair \( P \).
- \( \text{committed}(h) \) is the subsequence of \( h \) consisting of all events of committed transactions.
- \( \text{invalidated}(v) \) is the set of invalidated transactions in the commit event \( v \).
- \( \text{valid}(T_1, h) \) is true if \( \forall v, e \in \text{commits}(h) \rightarrow T_1 \notin \text{invalidated}(e) \), otherwise it is false.
- \( \text{active}(h) \) is the set of active (not committed or aborted) transactions.
- \( \text{value}(x, h) \) returns the associated value \( v \) from the last commit event (of the form, \( T \text{ commit, } I, W \)) where \( x \in \pi_1(W) \).
- \( \text{writes}(T_1, h) \) is the set of written variables (\( \pi_1(W) \)) of \( T_1 \)'s commit event in history \( h \).
- \( \text{reads}(T_1, h) \) is the set of variables in read events by \( T_1 \) in history \( h \).
- \( \text{length}(h) \) is the number of events in \( h \).
- \( \text{allowAbort}(T_1, T_2, h) \) is true if, in history \( h \), transaction \( T_1 \) is given permission to abort transaction \( T_2 \) by the contention manager, otherwise it is false.
- \( \text{invalid}(h) \) is true if \( h \) is accepted by a full invalidation automaton, otherwise it is false.

* This work was supported in part by Raytheon Company.

Justin E. Gottschlich, Jeremy G. Siek, and Manish Vachharajani
Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder
{gottschl, jeremy.siek, manishv}@colorado.edu

Proving Conflict Serializability for Full Invalidation

uctor Raytheon Company.
serial(h) is true if h is conflict serializable, otherwise it is false.

2.1 Types of Conflicts
Conflicts can be true or false. A true conflict is a conflict between two transactions such that the commit of both transactions may cause a history to be not serializable. A false conflict is one where the conflict cannot change the serializability of the history. Therefore, for a history h that is serializable, h remains serializable no matter how many committed, false conflicting transactions are appended to it.

Conflicts are true or false for a given system based on the precise definition of read and write events for that system. For this paper, we define conflicts as they exist in a lazy write acquisition system, where transactional writes are buffered until the transaction commits, and reads only see values from previously committed transactions, or their own, privately written value (see value(x, T1, h) above). In addition, our system uses lazy conflict detection and resolution [2], where conflicts are identified and resolved at a transaction’s commit-time. Using this model, the three possible types of conflicts, W-W, R-W, and W-R are defined as follows:

- W-W: transaction T1’s commit event writes to an element e and then another transaction T2’s subsequent commit event also writes to e. This type of conflict is false in our system because T1’s writes cannot be interleaved with T2’s writes no matter how they are executed. Instead, the write events happen atomically at commit-time, which binds the serializable execution order to the W-W commit order. In this case, T1 happens before T2, which is reflected in the commit order.

- W-R: transaction T2’s commit event writes to e and transaction T1 subsequently reads memory element e. This type of conflict is false in our system because the only possible value for T1 to see for e is one that is consistent T2’s committed value. In case, the serializable execution order is T2 happens before T1, which is reflected in the commit order.

- R-W: transaction T1 reads e and transaction T2’s subsequent commit event writes to e in history h. This type of conflict is true in our system, because T1 may have read a value that is inconsistent with the commit order. For example, if T2 commits at some point after T1, its read value of e will not reflect a T2 _<a T1 ordering. So, for any execution with a R-W to be serializable, T2 _<a T1. If T1 commits before T2’s ordering constraint is that T2 _<a T2; a serializable ordering. Yet, if T2 commits before T1 and another conflict exists between T1 and T2, which requires a T2 _<a T1 ordering, h will be not serializable.

2.2 Full Invalidation Automaton

We define a lazy conflict graph G(h) as a directed graph whose vertices are transactions and whose edges specify the necessary ordering of transactions. There is an edge (T1, T2) in G(h) if W-W(T1, T2, h), W-R(T1, T2, h), or R-W(T1, T2, h). A lazy conflict graph is named as such because it exploits optimizations of false conflicts that are present in a TM that employs lazy write acquisition (also known as deferred update [2]) and lazy conflict detection and resolution. Like Papadimitriou’s conflict graph [7], a

- Read
  Pre: \( T \in \text{active}(h) \land \text{valid}(T, h) \)
  Post: \( h' = h \cdot (T, x.\text{read}(<\text{value}(x, h)>)) \)

- Commit.1
  Pre:
  \[ T \in \text{active}(h) \land \text{valid}(T, h) \land (\forall T_x, T_z \in \text{active}(h) \land \text{valid}(T_z, h) \land (\text{writes}(T, h) \cap \text{reads}(T_z, h) \neq \emptyset) \rightarrow \text{allowAbort}(T, T_z)) \]
  Post:
  \[ 1. \ I = \emptyset \]
  \[ 2. \ T \in \text{active}(h) \land \text{valid}(T, h) \land (\forall T_x, T_z \in \text{active}(h) \land \text{valid}(T_x, h) \land (\text{writes}(T, h) \cap \text{reads}(T_z, h) \neq \emptyset) \land \text{allowAbort}(T, T_z)) \]
  \[ 3. \ h' = h \cdot (T, \text{commit}, I, W) \]

- Commit.2
  Pre:
  \[ T \in \text{active}(h) \land \text{valid}(T, h) \land (\exists T_x, T_z \in \text{active}(h) \land \text{valid}(T_x, h) \land (\text{writes}(T, h) \cap \text{reads}(T_z, h) \neq \emptyset) \land \text{allowAbort}(T, T_z)) \]
  Post: \( h' = h \cdot (T, \text{abort}) \)

Figure 1. Full Invalidation Automaton.

3. Full Invalidation Automaton
A full invalidation automaton is a concurrency control mechanism that accepts a sequence of instantaneous events known as a history or schedule. The automaton has an associated history, h, where h = 0 before accepting any event. When an event e is accepted by a full invalidation automaton, some corresponding event e0 is appended to the automaton’s history, such that h' = h . e0, where h represents the history prior to processing e and h' represents the history after processing it. A full invalidation TM automaton is defined by its preconditions and postconditions for the events it accepts, also known as its transition relation table [5], which is formally presented in Figure 1.

4. Theory
In this section we present conjectures that the previously defined automaton only accepts histories that are conflict serializable.
Conjecture 1. A history $h$ is conflict serializable if and only if its lazy conflict graph, $G(h)$, is acyclic.

Conjecture 2. Let $h = h_1 \cdot (T_1 \text{commit}, I_1, W_1) \cdot h_2$. If $\langle T_2, x.\text{read}(v_2) \rangle$ in $h_1$, then $\langle T_2 \text{commit}, I_2, W_2 \rangle \notin h_2$ when $x \in \pi_1(W_1)$.

Corollary 1. Commit events from both transactions $T_1$ and $T_2$ are not accepted by a full invalidation automaton if there is a RW conflict between $T_1$ and $T_2$.

Proof. Immediate from Lemma 2.

To show that full invalidation is conflict serializable, the general approach we plan to take is to show that if a full invalidation automaton only accepts schedules which are conflict serializable, then the full invalidation itself is conflict serializable. More specifically, this can be expressed in the following way: for all histories $h$, if $h$ is accepted by a full invalidation automaton, then $G(h)$ is acyclic, therefore $h$ is conflict serializable. Formally, this is expressed as follows.

Conjecture 3. For all $n$ and $h$, if length$(h) = n$ and inval$(h)$, then serial$(h)$.

5. Future Work

While we believe providing conjectures to prove a full invalidation automaton with read, write, commit, and abort events is a good first step to showing full invalidation is correct, we also believe several additional events should be modeled in order for our system to more closely model a real TM. We hope to extend the full invalidation model to include more complex events such as, the allocation and deallocation of memory elements, handling of user-level and system-level exceptions, and the arbitration of optimistic and pessimistic critical sections (i.e., lock-aware transactional memory). Furthermore, a key trait of transactions is their atomic behavior when one transaction is nested within another transaction, known as transactional composition or nested transactions [3, 5, 6]. Our future work also aims to include an analysis and extension of our automaton to support these types of transactions.

6. Conclusion

This paper presented a formalization our automaton, and a general outline of our definitions and the early conjectures that we plan to use to prove a full invalidation automaton only accepts histories that are conflict serializable. We believe that proving full invalidation is conflict serializable is important as it illustrates that efficient systems that employ full invalidation for their opacity and conflict detection mechanism such as, InvalSTM, produce serializable schedules.

References


