7-2016

Partial Aborts for Transactions via First Class Continuations

Matthew Le
ml9951@rit.edu

Rochester Institute of Technology

PARTIAL ABORTS FOR TRANSACTIONS VIA FIRST CLASS CONTINUATIONS

A THESIS SUBMITTED TO

THE FACULTY OF THE GOLISANO COLLEGE OF COMPUTER AND INFORMATION SCIENCES

IN CANDIDACY FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY

MATTHEW LE

ROCHESTER, NEW YORK

JULY 2016

Partial Aborts for Transactions via First Class Continuations

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Matthew Fluet, Supervisor

Dr. Arthur Nunes-Harwitt, Reader

Dr. Zack Butler, Observer

Table of Contents

List of Figures

ACKNOWLEDGMENTS

I would like to express an enormous amount of gratitude towards my advisor, Dr. Matthew Fluet. If it was not for him, the work presented in this thesis would unquestionably not have come to fruition. I would also like to thank Dr. Arthur Nunes-Harwitt for serving as the reader for this thesis and Dr. Zack Butler for serving as the observer. I am also eternally grateful for the love and support provided by family, without them I would most certainly not be where I am today.

ABSTRACT

Software transactional memory (STM) has proven to be a useful abstraction for developing concurrent applications where programmers denote transactions with an `atomic` construct that delimits a collection of reads and writes to shared mutable references. The runtime system then guarantees that all transactions are observed to execute atomically with respect to each other. Traditionally, when the runtime system detects that one transaction conflicts with another, it aborts one of the transactions and restarts its execution from the beginning. This can lead to problems with both execution time and throughput.

This thesis presents a novel approach that uses first-class continuations to restart a conflicting transaction at the point of a conflict, avoiding the re-execution of any work from the beginning of the transaction that has not been compromised. In practice, this allows transactions to complete more quickly, decreasing execution time and increasing throughput. The ideas presented in this thesis have been implemented in the context of the Manticore project, an ML-family language with support for parallelism and concurrency. Crucially, this work relies on constant-time continuation capturing via a continuation-passing-style (CPS) transformation and heap-allocated continuations. The partial abort scheme has been implemented as a part of three modern STM implementations: TL2, TinySTM, and NOrec. Experimental results show that, while no base STM implementation is universally best, each partial-abort implementation compares favorably to its full-abort counterpart.

In addition to an implementation, this thesis presents a formal semantics for partial aborts. A proof of correctness is given by relating the partial-abort semantics to an analogous full-abort semantics via a simulation. All proofs have been formally verified using the Coq Theorem Prover.

CHAPTER 1

INTRODUCTION

Software transactional memory (STM) [40, 25] allows programmers to mark sections of code as transactional using an `atomic` language construct (or using suitable library support). The runtime system then guarantees that modifications of shared references within transactions happen atomically with respect to other concurrently running transactions. Using STM instead of other synchronization methods such as mutex locks substantially simplifies the development of concurrent applications, avoiding common pitfalls such as deadlocks.

There are many different ways to enforce atomicity for STM. This work builds on multiple algorithms drawn from the STM literature. The general idea shared by all implementations is that when executing a transaction, the runtime system transparently maintains thread-local logs recording which references were read from and written to within a transaction. At the end of the transaction, the thread validates its log and if no conflicts are detected, it commits all of its writes to the global store. If a conflict is detected, then it throws away the logs and restarts the transaction from the beginning.

One issue that is under active research is that of fairness. Consider a situation where there are some threads executing long transactions and other threads that are executing short transactions that conflict with the long transactions. The threads executing the short transactions will complete sooner, giving them a higher probability of successfully validating and committing. These commits will then invalidate the long running transactions causing them to frequently abort. This issue has been addressed in the past by using contention managers [42], but not without imposing significant overheads.

In many compilers for functional languages, it is common to perform a continuation-passing-style (CPS) transformation to enable further optimizations. Additionally, it has been shown that continuations can be used to elegantly express concurrent programming [43, 41, 35] and serves as a fundamental component of the Manticore scheduling infrastructure [19]. This work makes use of

first-class continuations to restore execution of invalid transactions at the point of the first conflict, rather than always resuming execution at the beginning of the transaction. In practice, this avoids redundant work that has not been compromised by another thread, allowing threads to complete more quickly and increase throughput.

The idea of partially aborting transactions has been previously attempted in the context of C [26]. However, in order to capture a continuation in a non-CPS-converted language, the stack must be copied, which has linear complexity in both space and time. This makes capturing continuations at a fine granularity far too expensive. In order to deal with this, they require the programmer to manually insert "checkpoints," where continuations are to be captured. During the validation process, execution for aborted transactions returns to the latest valid checkpoint in the transaction. Even with manual checkpointing, the authors show a degradation in performance on both benchmarks presented due to the high overhead of stack copying.

When performing the CPS conversion of a program, each function is extended with an extra parameter called the return continuation. When the function finishes, rather than returning to a previous context on the stack, it invokes the return continuation with its result. This return continuation is often thought of as "the rest of the program," as it contains everything that is to happen next. The sort of checkpointing previously described can be implemented very efficiently by saving the return continuation when a transactional reference is read from and stored in the log. When a conflict is detected during validation, the program state can then simply be restored by invoking the continuation found in the log. What previously took linear time and space in a direct style language can now be done in constant time and space.

This thesis makes the following contributions:

- An extension of multiple STM implementations, including Transactional Locking II (TL2), TinySTM, and NOrec, to partially-abort transactions is presented.
- A significant overhead in garbage collection due to live captured continuations is identified and a scheme to bound the number of continuations held to a constant factor is presented.

- A formal semantics is given for both full-abort and partial-abort STMs and a machine checked proof using the Coq proof assistant is given. The proof establishes an equivalence between full-abort and partial-abort via a simulation.

- A detailed evaluation is presented covering a number of standard benchmarks common to the STM community. Results indicate that the overhead of capturing continuations to support partial aborts is negligible and can yield substantial performance improvements in certain contexts.

CHAPTER 2

BASELINE STMs

This chapter first describes the baseline full abort reference implementations that are later extended in Chapter 5. This thesis extends three different top performing STM implementations: TL2 [12], NOrec [9], and TinySTM [16]. All implementations support the following interface:

```
signature STM =
sig
    type 'a tref
    val new : 'a -> 'a tref
    val get : 'a tref -> 'a
    val put : 'a tref * 'a -> unit
    val atomic : (unit -> 'a) -> 'a
end
```

A `tref` is a transactional reference cell. This is akin to the traditional Standard ML reference cell, but with a transactional semantics. The `new` function is used for allocating a new `tref`, and `get` and `put` are used for reading and writing `tref`s respectively. The `atomic` function is used to evaluate a function transactionally, which then returns the result of the atomically executed function.

2.1 TL2

TL2 is one of the top performing implementations of STM and is commonly used in evaluating new STM algorithms [13, 4, 47]. The main novelty of TL2 is its use of a global version clock for eagerly detecting conflicts and ensuring atomicity. In this system, threads sample the global version clock at the beginning of each transaction. This version number is referred to as the read version for the transaction and is used for detecting `tref`s that have been altered since the start of

```
function put(tv, v):
  tls.writeSet.add(tv, v)
```

```
function get(tv):
  if(tv in tls.writeSet):
    return tls.writeSet[tv]
  v1 = tv.lock
  res = tv.contents
  v2 = tv.lock
  if(valid(v1) && v1 == v2):
    tls.readSet.add(tv)
    return res
  else
    abort()
```

Figure 2.1: TL2 Pseudocode for `put` and `get`

the transaction. Additionally, each `tref` has a version number / lock; the version number indicates when the `tref` was last updated.

TL2 buffers writes in a private write set recording both the `tref` to be written and the value to be written into it. When reading from a `tref`, the thread first consults its write set to check if it has already made updates to the `tref`, sometimes referred to as a read-after-write hazard. If so, it reads the value of its most recent update to the `tref` from its write set. If no local copy exists, then it checks that the version number associated with the `tref` is older than the read version it received at the beginning of the transaction and that the `tref`'s lock is not held. If these checks succeed, then it records the fact that it read from the `tref` in its read set. If the version number associated with the `tref` is newer than the read version or the `tref`'s lock is held, then the log is discarded and the transaction is aborted and restarted.

Figure 2.2 contains the pseudocode for committing a transaction. The thread first acquires the locks associated with each `tref` that it wrote. If any locks cannot be acquired, then the transaction is aborted in order to avoid deadlock. After all locks are acquired, the read set is validated by checking again that for each `tref` read, the version number associated with the `tref` is older than the read version received at the beginning of the transaction. If any are out of date, then the write locks are released and the transaction is aborted. If the read set is successfully validated, then an atomic increment of the global version clock is performed to retrieve a new version number that

```
function commit():
  foreach (tv, _) in tls.writeSet:
    if(!acquireLock(tv)):
      abort()
  foreach tv in tls.readSet:
    if(!valid(tv.lock)):
      releaseLocks()
      abort()
  write_version = fetch_and_add(VCLOCK, 1)
  foreach (tv, val) in tls.writeSet:
    tv.contents = val
    tv.lock = write_version
```

Figure 2.2: TL2 Pseudocode for `commit`

is referred to as the write version for the transaction. Lastly, for each local copy in the write set, the value is written to the corresponding `tref` and the write version is written into the version number associated with the `tref`, which releases the lock.

2.2 TinySTM

TinySTM [16] differs from TL2 in its approach to handling transactional writes, by using *encounter time locking*. In contrast to the private write buffering that TL2 does, TinySTM acquires a lock on the location being written as it is encountered in a transaction. Figure 2.3 contains the pseudocode for `put` and `get`. When a write is attempted, we first check to see if the writing transaction owns the lock. If so, then the `tref` can be written in place without any logging. If the lock is free and the version number is older than the transaction's read version, then the lock is acquired, the current value of the `tref` is logged in a private "undo log", and the new value is written in place. If lock acquisition fails then the lock was already owned by another transaction, or the version number is newer than the thread's read version and an abort occurs.

When a `tref` is read from, the thread first checks to see if it already owns the lock. If so, the value of the `tref` can be returned immediately. If the lock is not already owned, the thread checks that the lock is not acquired by another thread and that the corresponding version number is older

```
function put(tv, v):
  l = tv.lock
  if(l == tls.id | MSB):
    tv.contents = v
    return
  if(!valid(l)):
    abort()
  if(!CAS(tv.lock, l, tls.id | MSB)):
    abort()
  tls.writeSet.add(tv, tv.contents)
  tv.contents = v
```

```
function get(tv):
  v1 = tv.lock
  res = tv.contents
  if(v1 == tls.id | MSB):
     return res
  v2 = tv.lock
  if(valid(v1) && v1 == v2):
    tls.readSet.add(tv)
    return res
  else:
    abort()
```

Figure 2.3: TinySTM Pseudocode for `put` and `get`

than the thread's read version. If so, the value is returned, otherwise the transaction is aborted.

The commit protocol for TinySTM is the same as TL2 with the exception that locks do not need to be acquired since they are acquired inflight. The other difference is that when a thread needs to abort a transaction, the undo log is traversed, updating each `tref` with its corresponding backed-up value and releasing its lock. Livelock can be a serious problem for STMs with encounter time locking, so it is also common to add a randomized exponential backoff when aborting a transaction.

2.3 NOrec

NOrec [9] is an STM system optimized for low-cost, fast-path transactions. NOrec employs a single global lock which also doubles as a version clock. A single bit is designated for the lock and the remaining 63 bits are reserved for a version number.

When a thread begins a transaction, is fetches a read version by repeatedly sampling the version clock until the lock bit is clear. When writing a `tref`, the address of the `tref` and the value being written are privately buffered to a write set. When reading, the write set is scanned to check for read after write hazards. If the `tref` has not yet been written, the value is pulled out of the `tref` and the global clock is compared to the thread's read version. If they are equal, then the `tref` is added to the read set along with the value that it read, and the value is returned. If the global clock

```
function put(tv, v):
  tls.writeSet.add(tv, v)
```

```
function get(tv):
  if(tv in tls.writeSet):
    return tls.writeSet[tv]
  res = tv.contents
  while (tls.readVersion != VCLOCK):
    tls.readVersion =
      validate(tls.readSet)
    res = tv.contents
  tls.readSet.add(tv, res)
  return res
```

Figure 2.4: NOrec Pseudocode for `put` and `get`

```
function commit():
  while(!CAS(VCLOCK, tls.readVersion, tls.readVersion | MSB)):
    tls.readVersion = validate(tls.readSet)
  foreach (tv, val) in tls.writeSet:
    tv.contents = val
  VCLOCK = rv + 1
```

Figure 2.5: NOrec `commit`

is not equal to the thread's read version, then the read set is validated. If a `tref` is found to be out of date, then the transaction is aborted. If the entire read set is still valid, then the thread continues with the transaction with an updated read version equal to the global clock. This process continues until a valid snapshot is found or an abort takes place in `validate`.

To commit a transaction, the thread sets the designated lock bit of the read version and attempts to compare and swap this into the global version clock. If successful, the privately buffered writes are pushed into global view. If the compare and swap fails, then the read set is validated, either aborting or obtaining an updated read version. The compare and swap process is then repeatedly attempted until successful. Once the write set is written back, the global clock is incremented and the lock bit is cleared.

CHAPTER 3

SEMANTICS

partial-abort

This chapter first presents a formal semantics of the partial-abort extension for each respective STM and then proves that each extension is observably equivalent to its full abort analog.

3.1 Syntax

Figure 3 gives the syntax of the language. This language contains features which are not used by all STMs, but allows us to describe all three algorithms including their partial-abort counterparts using a single language. Values include lambda expressions, transactional reference locations, and the unit value. Expressions include values, variables, function application, transactional read, write, allocation, spawning threads, and atomic sections. Note that the **inatomic** expression form is an intermediate form denoting a running transaction and is not part of the surface language. Evaluation contexts are entirely conventional.

A heap is a mapping of transactional reference locations to values paired with a lock/version number. A lock can either be in an unlocked or locked state. If it is unlocked, it contains a version number, otherwise it contains a thread ID, a version number, and a backup value. For NOrec, the lock is unused and for TL2 only the version numbers are used. A thread pool is a collection of threads, where each thread maintains some transactional info. The transactional info can either be empty (denoted by $\cdot$), if the thread is not currently in a transaction, or be a triple containing the read version, a log, and the initial expression that the transaction is executing. Note that the initial expression is not used for the partial abort semantics, but is used for the full abort semantics. A transactional log contains three kinds of mappings. The first corresponds to a write, mapping a location to a value. Second is a checkpoint mapping corresponding to a read which maps a location to an evaluation context and the value that was read. Lastly is a non-checkpointed

$$
\begin{array}{rrcl}
\text{Version Numbers} & id, S, C & ::= & \mathbb{N} \\
\text{Values} & v & ::= & \lambda x.e \mid \ell \mid () \\
\text{Expressions} & e & ::= & v \mid x \mid e\ e \mid \textbf{spawn}\ e \\
 & & \mid & !e \mid e := e \mid \textbf{tref}\ e \\
 & & \mid & \textbf{atomic}\ e \mid \textbf{inatomic}(e) \\
\text{Evaluation Context} & \mathcal{E} & ::= & [\cdot] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \\
 & & \mid & !\mathcal{E} \mid \mathcal{E} := e \mid v := \mathcal{E} \mid \textbf{tref}\ \mathcal{E} \\
 & & \mid & \textbf{inatomic}(\mathcal{E}) \\
\text{Heap} & H & ::= & \cdot \mid H, \ell \mapsto (v, lock) \\
\text{Lock} & lock & ::= & (S)_u \mid (id, S, v)_l \\
\text{Thread Pool} & T & ::= & \cdot \mid \langle id; t; e \rangle \mid T \cup T \\
\text{Transaction Info} & t & ::= & \cdot \mid \langle S; L; e \rangle \\
\text{Log} & L & ::= & \cdot \mid L, \ell \mapsto_w v \mid L, \ell \mapsto_c (\mathcal{E}, v) \mid L, \ell \mapsto_r v
\end{array}
$$

Figure 3.1: Syntax

mapping corresponding to a read that cannot be checkpointed. As discussed later, the partial-abort TinySTM implementation should not create checkpoints after it has performed its first write.

3.2 Partial Abort Operational Semantics

In order to prove the correctness of performing partial aborts, the partial abort semantics is related to the original full-abort baseline semantics. Many rules are shared by the partial abort semantics and the full abort semantics (and an auxiliary replay semantics to be introduced in Section 3.4). To eliminate redundancy, the common rules have been factored out to a generic judgement denoted by $\rightarrow_{\text{mode, stm}}$ where "mode" is then instantiated by the appropriate full abort/partial abort mode and "stm" is instantiated to the appropriate STM (tl2, norec, or tiny).

The small-step operational semantics transitions one program state to another, where a program state consists of a monotonically increasing version clock, a heap, and a thread pool. The global version clock is used for both **tref** metadata and for generating unique thread identifiers. A source program e starts with the version clock set to 0, the empty heap, and a single thread $\langle id; \cdot; e \rangle$. A terminal program state consists of only threads that have finished evaluating their expressions to

$\boxed{C; H; T \rightarrow_{\text{mode, stm}} C'; H'; T'}$ mode $\in$ {full, partial, replay} stm $\in$ {tl2, norec, tiny}

$$\frac{C; H; T_1 \rightarrow_{\text{mode, stm}} C'; H'; T_1'}{C; H; T_1 \cup T_2 \rightarrow_{\text{mode, stm}} C'; H'; T_1' \cup T_2}\ \text{PARL} \qquad \frac{C; H; T_2 \rightarrow_{\text{mode, stm}} C'; H'; T_2'}{C; H; T_1 \cup T_2 \rightarrow_{\text{mode, stm}} C'; H'; T_1 \cup T_2'}\ \text{PARR}$$

$$\frac{T = \langle id; \cdot; \mathcal{E}[()]\rangle \cup \langle C; \cdot; e\rangle}{C; H; \langle id; \cdot; \mathcal{E}[\textbf{spawn}\ \ e]\rangle \rightarrow_{\text{mode, stm}} C + 1; H; T}\ \text{SPAWN}$$

$$\frac{}{C; H; \langle id; t; \mathcal{E}[(\lambda x.e)\ v]\rangle \rightarrow_{\text{mode, stm}} C; H; \langle id; t; \mathcal{E}[e[x \mapsto v]]\rangle}\ \text{BETA}$$

$$\frac{\ell \notin \text{Dom}(H)}{C; H; \langle id; \cdot; \mathcal{E}[\textbf{tref}\ v]\rangle \rightarrow_{\text{mode, stm}} C; H, \ell \mapsto (v, (0)_u); \langle id; \cdot; \mathcal{E}[\ell]\rangle}\ \text{ALLOC}$$

$$\frac{\text{read}_{\text{stm}}(H, L, \ell, id, S, \mathcal{E}) = (v, L')}{C; H; \langle id; \langle S; L; e_0\rangle; \mathcal{E}[!\ell]\rangle \rightarrow_{\text{mode, stm}} C; H; \langle id; \langle S; L'; e_0\rangle; \mathcal{E}[v]\rangle}\ \text{READ}$$

$$\frac{\text{write}_{\text{stm}}(H, L, \ell, id, S) = (H', L')}{C; H; \langle id; \langle S; L; e_0\rangle; \mathcal{E}[\ell := v]\rangle \rightarrow_{\text{mode, stm}} C; H'; \langle id; \langle S; L'; e_0\rangle; \mathcal{E}[()]\rangle}\ \text{WRITE}$$

$$\frac{e_0 = \mathcal{E}[\textbf{inatomic}(e)]}{C; H; \langle id; \cdot; \mathcal{E}[\textbf{atomic}\ e]\rangle \rightarrow_{\text{mode, stm}} C; H; \langle id; \langle C; \cdot; e_0\rangle; e_0\rangle}\ \text{ATOMIC}$$

$$\frac{}{C; H; \langle id; \langle S; L; e_0\rangle; \mathcal{E}[\textbf{atomic}\ e]\rangle \rightarrow_{\text{mode, stm}} C; H; \langle id; \langle S; L; e_0\rangle; \mathcal{E}[e]\rangle}\ \text{NATOMIC}$$

$$\frac{\textbf{validate}_{\text{stm}}(id; e_0; S; L; H; C + 1) \rightsquigarrow \textbf{commit}(\textbf{abort}(L', e', H_i), H_v)}{C; H; \langle id; \langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(v)]\rangle \rightarrow_{\text{mode, stm}} C + 1; H_v; \langle id; \cdot; \mathcal{E}[v]\rangle}\ \text{COMMIT}$$

$$\frac{\textbf{validate}_{\text{stm}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(L', e', H_i), H_v)}{C; H; \langle id; \langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(e)]\rangle \rightarrow_{\text{mode, stm}} C; H; \langle id; \langle C; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(e)]\rangle}\ \text{TSEXTEND}$$

Figure 3.2: Operational Semantics ($\rightarrow_{\text{mode, stm}}$: common rules)

values and are not in a transaction.

Rules PARL and PARR are used to nondeterministically choose a thread to execute. The SPAWN rule is used to create a new thread, where the newly created thread evaluates the expression given to **spawn**. In order to simplify the semantics, we do not allow threads to be created inside transactions. The BETA rule is used for applying a function, where $e[x \mapsto v]$ is the capture-avoiding substitution of v for x in e.

The READ rule is used for reading from a **tref**. This dispatches to an auxillary rule $\text{read}_{\text{stm}}(H, C, L, \ell, id, S, \mathcal{E}) = (v, L')$. This rule takes the heap, global clock, the thread's log, the location being read, the thread's id, read version, and evaluation context and outputs a value and a new log. The returned log (L') may either be the same log passed in (if the thread is reading from a **tref** it has already written to) or the log (L) extended with a new entry that is either a checkpointed or non-checkpointed read. The WRITE rule is used for writing to a **tref** within a transaction, and similarly dispatches to an auxiliary rule $\text{write}_{\text{stm}}(H, L, \ell, id, S) = (H', L')$, which encapsulates the writing semantics for each STM. It is worth noting that the READ and WRITE rules are partial in that the appropriate read version constraints need to be satisfied in read_{stm} and $\text{write}_{\text{stm}}$ respectively. In the event that the READ or WRITE rule is inapplicable, the TSEXTEND rule can be used to retrieve a newer timestamp (if the log is valid) or ABORT_PARTIAL/ABORT_FULL can be used to abort the transaction (to be introduced later).

The ALLOC rule creates a new reference, which can only be performed outside of a transaction. In the implementation, this restriction is not in place; however, this substantially simplifies the proof of equivalence discussed later.

The ATOMIC rule begins a transaction by grabbing a new read version from the global clock and transitioning into the **inatomic** intermediate form with transactional info initialized with the read version, an empty log, and the initial intermediate form. In our semantics, we do not allow nested transactions, so we treat them as idempotent (the NATOMIC rule). As noted in [26], partial aborts can be used to capture many common nested transaction idioms.

$\boxed{\text{read}_{\text{stm}}(H, C, L, \ell, id, S, \mathcal{E}) = (v, L') \qquad \text{stm} \in \{\text{tl2, norec, tiny}\}}$

$$\frac{L|_w(\ell) = v}{\text{read}_{\text{tl2}}(H, C, L, \ell, id, S, \mathcal{E}) = (v, L)}\ \text{TL2ReadLocal}$$

$$\frac{\ell \notin L|_w \quad H(\ell) = (v, (S')_u) \quad S \geq S'}{\text{read}_{\text{tl2}}(H, C, L, \ell, id, S, \mathcal{E}) = (v, L, \ell \mapsto_c (\mathcal{E}, v))}\ \text{TL2ReadGlobal}$$

$$\frac{L|_w(\ell) = v}{\text{read}_{\text{norec}}(H, C, L, \ell, id, S, \mathcal{E}) = (v, L)}\ \text{NOrecReadLocal}$$

$$\frac{\ell \notin L|_w \quad H(\ell) = (v, lock) \quad S = C}{\text{read}_{\text{norec}}(H, C, L, \ell, id, S, \mathcal{E}) = (v, L, \ell \mapsto_c (\mathcal{E}, v))}\ \text{NOrecReadGlobal}$$

$$\frac{H(\ell) = (v, lock) \quad \text{valid}(id, S, lock)}{\text{read}_{\text{tiny}}(H, C, L, \ell, id, S, \mathcal{E}) = (v, L, \ell \mapsto_r v)}\ \text{TinyRead}$$

$$\frac{H(\ell) = (v, lock) \quad \text{valid}(id, S, lock) \quad L|_w = \cdot}{\text{read}_{\text{tiny}}(H, C, L, \ell, id, S, \mathcal{E}) = (v, L, \ell \mapsto_c (\mathcal{E}, v))}\ \text{TinyReadChkpnt}$$

Figure 3.3: Read Rules

The COMMIT rule is used to commit a transaction. This rule relies on the **validate**$_{\text{stm}}$ judgement given in Figure 3.6; for now it suffices to know that if **validate**$_{\text{stm}}$ applied to a log L and current heap H yields **commit**(**abort**$(L', e', H_i), H_v)$, then the log could be validated in the current program state and H_v is the global heap with all locally written **tref**s committed. The COMMIT rule requires that **validate**$_{\text{stm}}$ yields a commit and then continues with the current heap replaced by the one returned by validation. The TSEXTEND rule validates the log mid-transaction. If the log is valid, then the thread is able to continue in the transaction with a read version equal to the current value of the global clock.

Figure 3.3 contains the STM specific rules for reading from a **tref**. The TL2ReadLocal rule is used for reading from a **tref** that has already been written by the thread in this transaction, where $L|_w$ is the log restricted to the write-mapped entries. This rule returns the value found in the log paired with the unchanged log. TL2ReadGlobal is used for reading directly from the **tref** and requires that the version associated with the **tref** is less than or equal to the thread's read version. This returns the value found in the **tref** and extends the log with a checkpointed read

$$\boxed{\text{write}_{\text{stm}}(H, L, \ell, v, id, S) = (H', L')} \qquad \text{stm} \in \{\text{tiny, tl2, norec}\}$$

$$\frac{}{\text{write}_{\text{tl2}}(H, L, \ell, v, id, S) = (H, L, \ell \mapsto_w v)}\ \text{TL2Write}$$

$$\frac{}{\text{write}_{\text{norec}}(H, L, \ell, v, id, S) = (H, L, \ell \mapsto_w v)}\ \text{NOrecWrite}$$

$$\frac{H(\ell) = (v', (id', S', v'')_l) \quad id = id'}{\text{write}_{\text{tiny}}(H, L, \ell, v, id, S) = (H, \ell \mapsto (v, (id, S', v'')_l), L)}\ \text{TinyWriteLocked}$$

$$\frac{H(\ell) = (v', (S')_u) \quad S \geq S'}{\text{write}_{\text{tiny}}(H, L, \ell, v, id, S) = (H, \ell \mapsto (v, (id, S', v')_l), L, \ell \mapsto_w v')}\ \text{TinyWriteUnlocked}$$

Figure 3.4: Write Rules

entry. NOrecReadLocal is used for reading from an already written **tref** and NOrecReadGlobal is used for reading directly from the **tref**, which requires that the thread's read version is equal to the global clock. TinySTM returns the value found in the heap as long as the associated lock is valid with respect to the thread's version number. The valid relation holds if the lock is in an unlocked state and the version is less than or equal to the read version, or if the lock is in a locked state and is owned by the thread. The TinyReadChkpnt rule extends the log with a checkpoint mapping if the log does not contain any write entries thus far. This restriction is elaborated later in Chapter 5.

Figure 3.4 contains the STM specific rules for writing to a **tref**. Writing to a **tref** is the same for TL2 and NOrec (TL2Write and NOrecWrite rules), where the log is simply extended with a write entry mapping the location to the value to be written. TinySTM is a bit more involved. The TinyWriteLocked rule is used for writing to a **tref** whose lock has already been acquired, as indicated by the thread's id appearing in the lock of the **tref** being written. The TinyWriteUnlocked rules is used for writing to a **tref** that has yet to be acquired. This rule requires that the **tref** is currently in an unlocked state and that the version number is less than or equal to the thread's read version.

Figure 3.5 contains the rules that distinguish the full abort from the partial abort semantics, which differ by only one rule. The ABORT_PARTIAL rule applies when the validate judgement

$\boxed{C; H; T \rightarrow_{\text{partial,stm}} C'; H'; T'}$

$$\frac{\textbf{validate}_{\text{stm}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H_i)}{C; H; \langle id; \langle S; L; e_0\rangle; e\rangle \rightarrow_{\text{partial,stm}} C; H_i; \langle id; \langle C; L'; e_0\rangle; e'\rangle} \text{ ABORT\_PARTIAL}$$

$\boxed{C; H; T \rightarrow_{\text{full,stm}} C'; H'; T'}$

$$\frac{\textbf{validate}_{\text{stm}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H_i)}{C; H; \langle id; \langle S; L; e_0\rangle; e\rangle \rightarrow_{\text{full,stm}} C; H_i; \langle id; \langle C; \cdot; e_0\rangle; e_0\rangle} \text{ ABORT\_FULL}$$

Figure 3.5: Partial and Full Abort Semantics

yields an abort. In this case, execution resumes with the log, expression, and heap returned by the validate judgement. The ABORT_FULL is analogous, but simply ignores the checkpoint information returned by the validate judgement and resumes execution with an empty log and the initial expression for the transaction.

3.3 Log Validation

In this section we present the validation rules for TinySTM. The validation semantics for NOrec and TL2 can be found in the Appendix, but are omitted here for brevity. Recall that TinySTM maintains an "undo log" by backing up values of **tref**s prior to writing for the first time. Thus, when committing a write, we simply change the associated lock to the unlocked state with a version number equal to the current value of the global clock. To abort a write, we replace the value of the **tref** with the backed up value and change the lock to the unlocked state with the backed up version number.

The validate judgement takes the thread id, initial expression, read version and log of the validating thread, along with the global heap and version clock. The result of validation is either an abort, which contains the log, expression, and heap to continue execution with, or a commit result. The commit result contains an abort result along with a heap H_v that has all of the thread's writes

$$\boxed{\textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(L'; e'; H_i); H_v) \mid \textbf{abort}(L'; e'; H_i)}$$

$$\frac{}{\textbf{validate}_{\text{tiny}}(id; e_0; S; \cdot; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(\cdot; e_0; H); H)}\ \text{CEMPTY}$$

$$\frac{H(l) = (v, (id, S', v')_l) \quad \textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H_i)}{\textbf{validate}_{\text{tiny}}(id; e_0; S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H_i, \ell \mapsto (v', (S')_u))}\ \text{APWRITE}$$

$$\frac{\textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H_i)}{\textbf{validate}_{\text{tiny}}(id; e_0; S; L, \ell \mapsto_r v; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H_i)}\ \text{APREAD}$$

$$\frac{\textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H_i)}{\textbf{validate}_{\text{tiny}}(id; e_0; S; L, \ell \mapsto_c (\mathcal{E}, v); H; C) \rightsquigarrow \textbf{abort}(L'; e'; H_i)}\ \text{APCHKPNT}$$

$$\frac{\textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(L'; e'; H_i); H_v) \quad H(l) = (v, (id, S', v')_l)}{\textbf{validate}_{\text{tiny}}(id; e_0; S; L, \ell \mapsto_w v'; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(L'; e'; H_i, \ell \mapsto (v', (S')_u)); H_v, \ell \mapsto (v, (C)_u))}\ \text{CPWRITE}$$

$$\frac{\textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v, lock) \quad \text{valid}(id, S, lock)}{\textbf{validate}_{\text{tiny}}(id; e_0; S; L, \ell \mapsto_c, (\mathcal{E}, v'); H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(L; \mathcal{E}[!\ell]; H_i); H_v)}\ \text{CPCHKPNT}$$

$$\frac{\textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v, lock) \quad \text{valid}(id, S, lock)}{\textbf{validate}_{\text{tiny}}(id; e_0; S; L, \ell \mapsto_r v'; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v)}\ \text{CPREAD}$$

$$\frac{\textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v, lock) \quad \text{invalid}(id, S, lock)}{\textbf{validate}_{\text{tiny}}(id; e_0; S; L, \ell \mapsto_c (\mathcal{E}, v'); H; C) \rightsquigarrow \textbf{abort}(L; \mathcal{E}[!\ell]; H_i)}\ \text{ACHKPNT}$$

$$\frac{\textbf{validate}_{\text{tiny}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v, lock) \quad \text{invalid}(id, S, lock)}{\textbf{validate}_{\text{tiny}}(id; e_0; S; L, \ell \mapsto_r v'; H; C) \rightsquigarrow \text{chkpnt}}\ \text{AREAD}$$

Figure 3.6: TinySTM Transactional Log Validation

committed. The reason for containing an abort result within a commit result is to handle the situation where some portion of the log containing at least one write has successfully been validated, and then a later entry in the log is found to be invalid. In this case, we need to "retroactively undo" the earlier writes. To facilitate this, when the log is valid we maintain enough information to either propagate the commit or abort the transaction.

Figure 3.6 gives the rules for validating a transactional log. The CEMPTY rule indicates that the empty log can trivially be validated. The log and expression associated with the commit are the empty log and initial expression respectively. Thus, if the log is found to be invalid before a valid checkpoint entry is encountered, the result is equivalent to a full abort. The APWRITE, APREAD, and APCHKPNT rules propagate an abort through a write, read, or checkpoint mapping in the log: if validation failed on an earlier operation in the log, then the entire validation process

aborts; note that this propagates the earliest conflict information. The CPWRITE rule propagates a commit through a write mapped entry, by extending the abort heap with a mapping to the backed up value paired with the backed up version in the unlocked state. Additionally, it extends the commit heap with a mapping to the current value of the **tref** paired with the thread's write version in the unlocked state. The CPCHKPNT and CPREAD rules propagate a commit through a valid read by dispatching to the valid judgement, which for TinySTM requires that either the thread owns the lock, or the lock is not held and the associated version number is less than or equal to the thread's read version (S). Note that the CPCHKPNT rule propagates the current log and builds an expression out of the current location and evaluation context, whereas CPREAD propagates the checkpoint information from the recursive invocation. Lastly, the ACHKPNT and AREAD rules initiate an abort for an invalid read. The only difference between these two rules is that ACHKPNT returns an abort result with the current log and builds an expression from the evaluation context stored in the log entry, whereas AREAD simply uses the log and expression returned from the recursive invocation.

3.4 Equivalence

The correctness of the various full abort algorithms have been discussed in previous works [27] [9] [16]. In this paper, we simply prove that performing partial aborts yields the same final program states as performing full aborts and use this equivalence to deduce the correctness of our extension. There are two directions of this proof, one which shows that full abort can simulate partial abort and another which shows that partial abort can simulate full abort.

The intuition behind the first direction is that if the partial abort semantics admitted some non-atomic trace for some arbitrary program, then we would have a proof that using the full abort semantics we could yield an equivalent trace, which would imply that the full abort semantics also admits non-atomic traces. This, however, would contradict the correctness of the full-abort semantics, which we are assuming to be true. The importance of the second direction (partial abort

$\boxed{\text{WellFormed}_{\text{stm}}(C; H; T) \qquad \text{stm} \in \{\text{tl2, norec, tiny}\}}$

$$\frac{}{\text{WellFormed}_{\text{stm}}(C; H; \langle id; \cdot; e\rangle)}$$

$$\frac{C; H; \langle id; \langle S; \cdot; e_0\rangle; e_0\rangle \rightarrow^*_{\text{replay,stm}} C; H; \langle id; \langle S; L; e_0\rangle; e\rangle}{\text{WellFormed}_{\text{stm}}(C; H; \langle id; \langle S; L; e_0\rangle; e\rangle)}$$

$$\frac{\text{WellFormed}_{\text{stm}}(C; H; T_1) \quad \text{WellFormed}_{\text{stm}}(C; H; T_2)}{\text{WellFormed}_{\text{stm}}(C; H; T_1 \cup T_2)}$$

Figure 3.7: Thread Pool WellFormed Judgement

can simulate full abort) is to show that our partial abort semantics is not vacuous. Clearly, the empty set of traces can be simulated by full abort, but is not a useful language. This direction of the proof shows that the partial abort semantics maintains the full power of the full abort semantics.

In order to show that our partial abort extension has the same desirable properties as the full abort algorithm, we prove the following theorem:

Theorem 1 (Equivalence)**.** $\forall e\ C\ H\ T$ stm*, if* $\text{Done}(T)$*, then* $0; \cdot; \langle id; \cdot; e\rangle \rightarrow^*_{\text{partial, stm}} C; H; T$ *iff* $0; \cdot; \langle id; \cdot; e\rangle \rightarrow^*_{\text{full, stm}} C; H; T$.

where $\text{Done}(T)$ specifies that every thread in T is not in a transaction and has evaluated its expression to a final value. The proof proceeds by proving the two directions of the if and only if.

First, we give a well-formedness judgement in Figure 3.7. This essentially says that for each thread currently in a transaction, the transaction can be re-executed from the beginning to its current state using a "replay" semantics, which consists of the common rules ($\rightarrow_{\text{mode, stm}}$) and one additional $\rightarrow_{\text{replay,stm}}$ rule (Figure 3.8). The READ_REPLAY rule allows a thread to read from an out-of-date **tref** with a few restrictions:

- If the **tref** is locked by another thread, but the backed up version number is still valid, then the thread reads the backed-up value. This way if the thread that owns the lock aborts, the replay derivation can be reproduced but using the regular READ rule instead

$\boxed{C; H; T \rightarrow_{\text{replay,stm}} C'; H'; T'}$ stm $\in$ {tl2, norec, tiny}

$$\frac{H(\ell) = (v, lock) \quad \text{mkVal}(lock) = v'}{C; H; \langle id; \langle S; L; e_0 \rangle; \mathcal{E}[!\ell] \rangle \rightarrow_{\text{replay,stm}} C; H; \langle id; \langle S; L, \ell \mapsto_r \mathcal{E}; e_0 \rangle; \mathcal{E}[v'] \rangle} \text{ READ\_REPLAY}$$

$\boxed{\text{mkVal}(id, S, lock) = v}$

$$\frac{S \geq S' \quad id \neq id'}{\text{mkVal}(id, S, (id', S', v)_l) = v} \text{ mkOldVal} \qquad \frac{S < S' \quad id \neq id'}{\text{mkVal}(id, S, (id', S', v)_l) = v'} \text{ mkLocked}$$

$$\frac{S < S'}{\text{mkVal}(id, S, (S')_u) = v'} \text{ mkUnlocked}$$

Figure 3.8: Replay Semantics

- If the **tref** is locked by another thread and the backed up version number is greater than the thread's read version, then the thread is able to "pull a value out of thin air." If the thread that owns the lock aborts, the replay derivation will still need to use READ_REPLAY
- If the **tref** is unlocked, but the version number is greater than the thread's read version, then a value can be pulled out of thin air.

This rule makes it easy to show that well-formedness is preserved by the partial abort step relation ($\rightarrow_{\text{partial,stm}}$); in particular, when one thread commits via the COMMIT rule, other threads' logs may become invalid due to the updates to the global heap, yet they remain replay-able via the READ_REPLAY rule. With the WellFormed judgement, we can prove the forward direction of Theorem 1 using the following theorem:

Theorem 2 (Partial Implies Full)**.** $\forall C\ C'\ H\ H'\ T\ T'$ stm,
if $\text{WellFormed}_{\text{stm}}(C; H; T)$ *and* $C; H; T \rightarrow^*_{\text{partial, stm}} C'; H'; T'$,
then $C; H; T \rightarrow^*_{\text{full, stm}} C'; H'; T'$.

Proof Sketch. By induction on the derivation of $C; H; T \rightarrow^*_{\text{partial, stm}} C'; H'; T'$ and case analysis on the first $\rightarrow_{\text{partial, stm}}$ step taken. The interesting case is the ABORT_PARTIAL rule. In this case,

$\boxed{\text{AheadOf}_{\text{stm}}(C; H; T) \qquad \text{stm} \in \{\text{tl2, norec, tiny}\}}$

$$\frac{}{\text{AheadOf}_{\text{stm}}(C; H; \langle id; \cdot; e\rangle; \langle id; \cdot; e\rangle)}$$

$$\frac{C; H; \langle id; \langle S; L; e_0\rangle; e\rangle \rightarrow^*_{\text{replay,stm}} C; H; \langle id; \langle S; L'; e_0\rangle; e'\rangle}{\text{AheadOf}_{\text{stm}}(C; H; \langle id; \langle S; L; e_0\rangle; e\rangle; \langle id; \langle S; L'; e_0\rangle; e'\rangle)}$$

$$\frac{\text{AheadOf}_{\text{stm}}(C; H; T_{\text{f1}}; T_{\text{p1}}) \quad \text{AheadOf}_{\text{stm}}(C; H; T_{\text{f2}}; T_{\text{p2}})}{\text{AheadOf}_{\text{stm}}(C; H; T_{\text{f1}} \cup T_{\text{f2}}; T_{\text{p1}} \cup T_{\text{p2}})}$$

Figure 3.9: Thread Pool AheadOf Judgement

partial abort steps to the thread state returned from the **validate** judgement and full abort steps to the initial expression recorded at the beginning of the transaction. We need to show that full abort can "catch up" to partial abort, which can be done by simulating the derivation provided by well-formedness. Note that the replay of the aborted thread does not require the READ_REPLAY rule, since the partially-aborted thread has been restarted with a valid log.

The remaining cases simply make use of the induction hypothesis. Note that in order to make use of the induction hypothesis, we must show that the state stepped to is also well formed, which we prove in Theorem 4 □

The other direction of the proof is slightly trickier. The problem is that we need to show that if a full abort takes place, then there is an equivalent partial abort step. Basically, we need a way of specifying that the partial abort program state is in a "possible future state" of the full abort program state. To do so, we give an "ahead-of" judgement in Figure 3.9 that relates two thread pools. The AheadOf relation specifies that a transactional thread in one pool is related to a corresponding transactional thread in the other pool if the first thread can "catch up" to the second thread using the replay step relation ($\rightarrow_{\text{replay,stm}}$) and specifies that a non-transactional thread is only related to an identical non-transactional thread. Therefore, if $\text{AheadOf}_{\text{stm}}(C; H; T_{\text{f}}; T_{\text{p}})$ and T_{f} is either the initial program state or a final program state, then it must be the case that $T_{\text{p}} = T_{\text{f}}$. With the AheadOf judgement, we can prove the backward direction of Theorem 1 using the following

theorem:

Theorem 3 (Full Implies Partial)**.** $\forall C\ C'\ H\ H'\ T_p\ T_p'\ T_f\ T_f'$ stm,

if $\text{AheadOf}_{\text{stm}}(C; H; T_f; T_p)$ *and* $\text{WellFormed}_{\text{stm}}(C; H; T_p)$ *and* $C; H; T_f \rightarrow^*_{\text{full, stm}} C'; H'; T_f'$,

then $C; H; T_p \rightarrow^*_{\text{partial, stm}} C'; H'; T_p'$ *and* $\text{AheadOf}_{\text{stm}}(C'; H'; T_f'; T_p')$.

Proof Sketch. By induction on $C; H; T_f \rightarrow^*_{\text{full, stm}} C'; H'; T_f'$ and case analysis on the first $\rightarrow_{\text{full, stm}}$ step taken. We then also do case analysis on the $\text{AheadOf}_{\text{stm}}(C; H; T_f; T_p)$ derivation, which states that T_f can take zero or more replay steps to get to T_p. If the number of steps taken is nonzero, then we do not need to do anything with T_p as it is still "in the future" of T_f. If zero replay steps were taken to get from T_f to T_p, then $T_f = T_p$ and we must simulate the full abort step taken. The most interesting cases are the COMMIT and ABORT_FULL rules.

Case COMMIT: In this case, we know that the full abort thread is ready to commit, so it must be of the form $\langle\langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(v)]\rangle$. From $\text{AheadOf}_{\text{stm}}(C; H; \langle\langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(v)]\rangle; T_p)$, we know that T_p is of the form: $\langle\langle S; L'; e_0\rangle; e'\rangle$ and

$C; H; \langle\langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(v)]\rangle \rightarrow^*_{\text{replay,stm}} C; H; \langle\langle S; L'; e_0\rangle; e'\rangle$, but there is no way for this thread to take a step while remaining in the transaction, since it has finished evaluating its expression to a value. Therefore, it must be the case that $T_p = \langle id; \langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(v)]\rangle$, allowing it to also commit in the partial abort semantics.

Case ABORT_FULL: In this case, full abort returns to the beginning of its transaction, while partial abort is going to resume at some intermediate point. We must show that the AheadOf relation still holds for whatever intermediate point partial abort returns to. By the well formedness of the partial abort thread pool, we know that we can replay the derivation from it's beginning to it's current position. We can then construct the AheadOf derivation by simulating this replay derivation up to the point that partial abort returns to as determined by the validate judgement.

□

The remaining critical component of the proof is showing that the partial abort step relation preserves the well formedness property.

Theorem 4 (Step Preserves Well Formedness)**.** $\forall C\ C'\ H\ H'\ T\ T'$ stm*,*

if $C; H; T \rightarrow_{\text{partial,stm}} C'; H'; T'$ *and* $\text{WellFormed}_{\text{stm}}(C; H; T)$*, then* $\text{WellFormed}_{\text{stm}}(C'; H'; T')$

The proofs for TL2 and NOrec are quite similar; however, since TinySTM modifies the heap in place for in-flight transactions the proof ends up being a bit more involved.

TL2/NOrec Proof Sketch. By induction on $C; H; T \rightarrow_{\text{partial,stm}} C'; H'; T'$. The interesting cases are PARL and PARR, where the thread pool is split in two subpools and a step is taken on one of the subpools. In these cases we must show that one subpool's step does not interfere with any thread's well formedness in the complementary (stationary) subpool.

The only rules that could potentially interfere with a thread's well formedness are the ALLOC, ATOMIC, and COMMIT rules as all other rules leave the heap and version clock alone, so they could not possibly compromise any other thread's well formedness.

Case ALLOC: For the ALLOC case, the heap is extended with a single extra binding mapping a fresh location ℓ to the initialization value paired with a version equal to 0. We now need to show that the stationary subpool (T_2 for PARL and T_1 for PARR) is still well formed with the heap extended with this newly allocated **tref**. For each thread in the stationary subpool, we must show that it can still replay from the beginning of its transaction to where it is now. It must be the case that no step in the previous derivation changes, since it is not possible that any thread in the stationary subpool could have read or written to the newly allocated **tref**, since by definition it was not previously in the heap.

Case ATOMIC: For the ATOMIC case we need to show that any thread in the stationary subpool is still well formed with a version clock value equal to $C + 1$. This is trivial to show since none of the steps in the replay derivation of well formedness rely on the value of the version clock.

Case COMMIT: For the COMMIT case, the value of the resulting version clock will be incremented by one, and there may be an arbitrary number of writes added to the front of the heap. We must show that for each thread in the stationary subpool it can still replay from the beginning to its current position with the extended heap. This relies on Lemma 5, which intuitively states that

if a thread can replay from one state to another, then it can also replay from those two points with a heap extended by some number of entries, such that every entry in the extension has a version equal to the value of the global clock plus one.

□

TinySTM Proof Sketch. By induction on $C; H; T \rightarrow_{\text{partial,stm}} C'; H'; T'$ and case analysis on the first $\rightarrow_{\text{partial,stm}}$ step taken. The TinySTM proof follows a similar structure as the TL2/NOrec proof in that the most interesting cases are the PARL and PARR cases and we must show that one thread's step does not interfere with a stationary thread's well formedness. The ALLOC, ATOMIC, and COMMIT cases are analogous to the TL2/NOrec proof, however, we must also consider the WRITE and ABORT_PARTIAL cases since WRITE modifies the global heap in place and ABORT_PARTIAL may unlock some **tref**s.

Case WRITE: In this case, we must show that if one thread takes a WRITE step, other threads can still replay their transactions. After a thread makes use of the WRITE rule, the **tref** being written is in the locked state. We must show that other threads are still able to replay their transactions despite this. This is proven in another lemma analogous to Lemma 5. Intuitively, it holds because a thread can use the READ_REPLAY rule if it previously read from the newly written **tref**. In this case, it will continue with the transaction using the value determined by either the mkOldVal or mkLocked rules.

Case ABORT_PARTIAL: This case is similar to the previous in that we also make use of an auxillary lemma that says a thread can still replay its transaction if another thread aborts its transaction, in which case some **tref**s which were previously locked may now be unlocked. Again, the interesting case to consider is when replaying READ/READ_REPLAY rules. If an aborted transaction ends up unlocking a **tref** read from by the replaying transaction, then it can now either replay using READ_REPLAY (and mkUnlocked) if the backed up version is newer than the thread's read version, or it can use the normal READ rule if the backed up version is less than or equal to the thread's read version.

□

Lemma 5 (Replay Heap Append)**.** $\forall C\ H\ H'\ S\ e_0\ L\ e$ stm*,*

if $C; H; \langle id; \langle S; \cdot; e_0\rangle; e_0\rangle \rightarrow_{\text{replay,stm}} C; H; \langle id; \langle S; L; e_0\rangle; e\rangle$ *and* $\forall(\ell \mapsto (v, lock)) \in H', lock = (C+1)_u$*,*

then $C; H +\!\!+ H'; \langle id; \langle S; \cdot; e_0\rangle; e_0\rangle \rightarrow_{\text{replay,stm}} C; H +\!\!+ H'; \langle id; \langle S; L; e_0\rangle; e\rangle$

Proof Sketch. By induction on $C; H; \langle id; \langle S; \cdot; e_0\rangle; e_0\rangle \rightarrow_{\text{replay,stm}} C; H; \langle id; \langle S; L; e_0\rangle; e\rangle$ and case analysis on the first step taken in the derivation. The only interesting case is READ. It must be the case that when looking up the **tref** being read from in $H +\!\!+ H'$, either we get the same value returned in the READ step of the original derivation, or we get back a value that was added in H'. In this case, we know that the associated version is strictly greater than the thread's read version, in which case we can use the READ_REPLAY rule allowing us to "make up a value" to continue with, in which case we choose to continue with the same value used in the READ of the previous derivation. □

The full details of all of the above proofs can be found in the Coq formalization at `https://github.com/ml9951/ICFP15-Coq-Proofs`.

CHAPTER 4

MANTICORE

The partial-abort extension has been implemented in the context of the Manticore project [18]. Manticore is an effort to design and implement a functional programming language with support for parallelism and concurrency. It consists of: the *Parallel ML (PML)* language, a parallel dialect of Standard ML [29] extended with implicit fine-grain parallelism [20] and with explicit CML-style concurrency [37, 35]; the `pmlc` compiler, a whole-program compiler from PML source to native x86-64 (*a.k.a.*, AMD64) code; and the *Manticore runtime system*, which provides memory management, process abstraction, thread scheduling, work stealing, and message passing. In this section, we highlight a few details about the compiler and runtime system that are relevant to the implementation of partial-abort transactions in Manticore.

4.1 Compiler Architecture

The `pmlc` compiler is a whole-program compiler and has the standard organization as a sequence of transformations between and optimizations of various intermediate representations (IRs). There are six distinct IRs in the `pmlc` compiler:

1. Parse tree - the result of parsing
2. AST - an explicitly-typed abstract syntax tree representation, produced by type checking
3. BOM - a direct-style normalized λ-calculus
4. CPS - a continuation-passing-style λ-calculus
5. CFG - a first-order control-flow graph representation
6. MLTree - an expression-tree representation used by the MLRISC code-generation framework [21]

4.1.1 BOM

The BOM IR plays a key role in the implementation of the Manticore runtime system. Although a small runtime kernel that implements garbage collection (see Section 4.2) and various machine-level scheduler operations are written in C, the majority of the Manticore runtime system, including the scheduling infrastructure [19] and the STM implementation of this work, is written in (an unnormalized, external, concrete sytax for) BOM.[1] In order to compile a program, the `pmlc` compiler loads both PML source code written by the user and BOM runtime code written by the developers. By defining much of the runtime system in external files in BOM, it is easy to modify the implementation of many aspects of the runtime system in an expressive language with higher-order functions, pattern matching, and garbage collection. Furthermore, since BOM is a compiler IR, the user application code and the runtime system code can be combined and optimized together.

The BOM IR has several notable features:

- It supports first-class continuations with a binding form that reifies the current continuation. First-class continuations are a well-known language-level mechanism for expressing concurrency [43, 24, 36, 33, 41, 37]; they serve as the foundation for the Manticore scheduling infrastructure [19] and are used in this work for efficiently performing partial aborts of transactions.

- It supports mutable tuples, whereby individual fields of the tuple may be mutated in place. (In PML, tuples are immutable and mutable references necessarily incur a level of indirection.)

- It includes atomic operations, such as *compare-and-swap*.

1. Technically, the `pmlc` compiler allows inline BOM, similar in spirit to inline assembly, to be embedded in PML source files; this is the mechanism by which features implemented in BOM are made available in the surface language.

4.1.2 CPS, CFG, and Heap-Allocated Continuations

The CPS IR is the final higher-order representation used in the compiler. For the translation from the BOM IR to the CPS IR, the Danvy-Filinski CPS transformation [11] is used, but the implementation is simplified by the fact that BOM is a normalized direct-style representation. The translation from direct style to continuation-passing style eliminates the special handling of continuations, so that capturing a continuation is effectively a variable-variable copy and subject to copy propagation, and makes control flow explicit. Using higher-order control-flow analysis, we perform a number of further optimizations on the CPS IR program, such as arity-raising [6] and aggressive inlining [5].

The CPS IR is translated to the CFG IR, a first-order control-flow-graph representation, by applying closure conversion. The transformation also handles the heap allocation of first-class continuations *à la* SML/NJ [2]. Although heap-allocated continuations impose some extra overhead for sequential execution, due to a high allocation rate of short-lived data and more frequent garbage collections, they provide a number of advantages:

- Creating/capturing a continuation just requires the heap allocation of a small (< 100 bytes) object, so it is fast and imposes little space overhead.
- Since continuations are *immutable values*, many nasty race conditions in the scheduler can be avoided.
- Heap-allocated first-class continuations do not have the lifetime limitations of one-shot [7] and escaping [34, 17] continuations, which is essential for the work presented here.

4.2 Garbage Collection and Heap Architecture

The Manticore garbage collector is based on a novel combination of the Doligez-Leroy-Gonthier (DLG) parallel collector [15, 14] and the Appel semi-generational collector [1] and is described

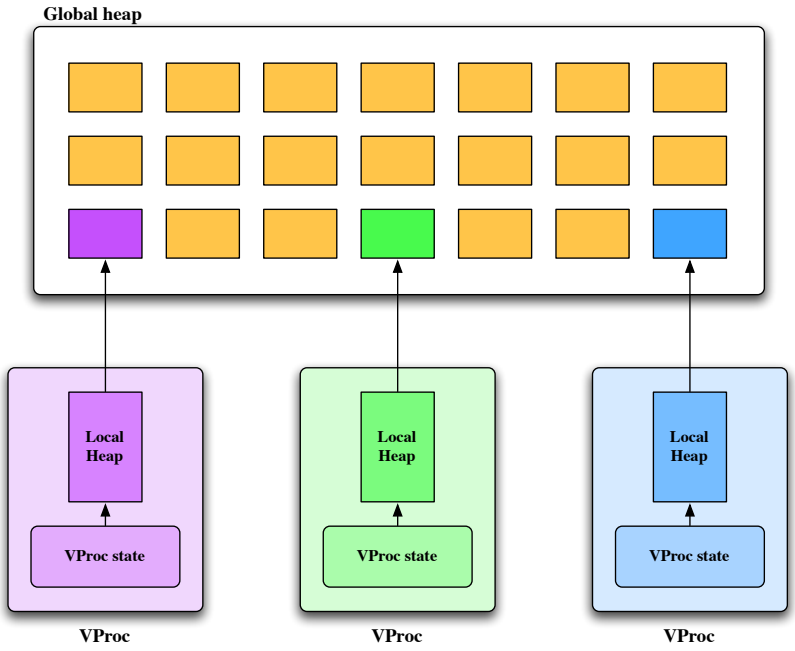

Figure 4.1: Manticore Heap Architecture

more fully in previous work [3]. From the DLG collector, we adopt an overall heap architecture with both a private local heap for each *virtual processor* (an abstraction of a hardware processor) and a global heap shared by all virtual processors; the Appel collector is used to garbage collect the local heaps. Threads executing on a virtual processor allocate new data in the virtual processor's local heap. When the local heap is full, a minor collection is performed and, if necessary, a major collection promotes live data from the local heap to the global heap. So that minor and major collections of a virtual processor can be completed without synchronizing with other virtual processors to establish a root set, we adopt two invariants from the DLG collector: first, there cannot be any pointers from the global heap into any local heap, and, second, there cannot be any pointers from one local heap into another local heap. In order to maintain these invariants, it is occasionally necessary to explicitly promote newly allocated data to the global heap in order to pass a reference to the data to another virtual processor or to update a mutable object in the global heap to reference the data.

CHAPTER 5

IMPLEMENTATION

The various STM libraries are implemented in the BOM IR, which as previously mentioned includes mutable references and first-class continuations. This substantially simplifies the implementation, requiring zero modifications to the compiler or runtime kernel. Source code for our implementation can be found at `http://manticore.cs.uchicago.edu`.

In BOM, a tref can be represented as:

```
type 'a tref = !('a * long * long)
```

where the `!` indicates that the type is a mutable tuple. The first element of the tuple is for the contents of the `tref` that are read from and written to by the programmer. The second and third elements correspond the current lock value and the previous lock value respectively. We designate the most significant bit for the lock, and the remaining 63 bits for the version number. This way, a thread can check if a `tref` is locked or "too new" with a single greater-than comparison. When acquiring a lock, a thread uses its unique threadID with the most significant bit set, i.e. `((1 << 63)|threadID)`.

Each thread maintains four pieces of information within its thread local storage: a thread ID, a read version, a write set, and a read set. When a thread begins executing a transaction, it acquires the read version from the global clock. For NOrec, the version clock doubles as a commit lock. When beginning a transaction, it must repeatedly sample the version clock until it receives a value corresponding to an unlocked state.

5.1 Reads

For TL2, each time a `tref` is read, the write set is first consulted to determine if the `tref` has been written to during the transaction; if so, then the value of the most recent entry for the `tref` in

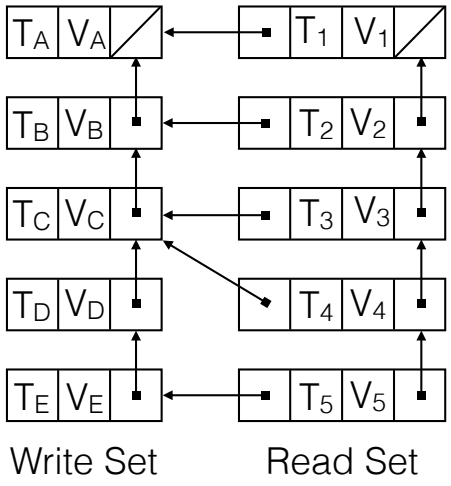

Figure 5.1: Layout of Read/Write sets

the write set is returned. If there is no entry for the tref in the write set, then the tref is checked for validity, by comparing the version number associated with the `tref` to the thread's read version. If the `tref` is valid, then an entry is added to the read set that records the `tref` being read, the current continuation, and a pointer to the current write set as depicted in Figure 5.1. If the `tref` is out of date, then we acquire a version number from the global clock and validate the read set as described in Section 5.4, which will determine if a partial abort is necessary.

NOrec similarly checks the write set first for previous modifications, returning the current value if found. If the `tref` is not found in the write set, then the `tref` is dereferenced. The global version clock is then compared to the thread's read version. If they are the same, then the value of the `tref` can be returned and an entry containing the `tref`, value retrieved from `tref`, and pointer to current write set can be added to the read set. If the read version is not equal to the version clock, then the read set is validated. If the entire read set is valid, then the process is repeated; if not, the transaction is aborted.

TinySTM does not consult the write set on a read, instead it checks the validity of the lock. If it owns the lock, then the value stored in the `tref` can be returned. If not, then we check that the lock is not held and that the version number is less than or equal to the thread's read version.

Currently, we do not allow TinySTM to create a checkpoint *after* a write has been performed. To understand why, consider the following example:

Time	Thread 1	Thread 2
1	`x := 1`	`y := 1`
2	`!z`	`!z`
3	`!y`	`!x`

At timestep 1, both threads simultaneously write to distinct `tref`s, `x` and `y`. At timestep 2, they both read from a third `tref` (`z`) and then in timestep 3, read from the `tref`s written at timestep 1. At this point, each thread will see that the locks have been acquired by another thread and an abort needs to take place. If we create a checkpoint at timestep 2, execution will resume after timestep 2. This will cause the locks for `x` and `y` to never be released and the program will enter a livelock. On the other hand, if a full abort took place, `x` and `y` would get unlocked each time the transaction was aborted, giving the opportunity for progress to be made.

To avoid this problem, we simply do not allow TinySTM to create checkpoints after a write has occurred. We do not believe this to be a severe limitation as it has been pointed out in the past that transactions are commonly written in a read-mostly planning phase followed by a write-mostly completion phase, which forms the basis of transaction partitioning [46][45].

5.2 Writes

For TL2 and NOrec, writing to a `tref` simply amounts to adding the `tref` and value to be written to the thread local write set. Each time a write is performed, an entry is added to the write set non-destructively. This is crucial for performing partial aborts, and also does not violate the generational garbage collector invariants.

For TinySTM, writes are a bit more involved. The thread first checks to see if it already owns the lock for the `tref` it is writing. If so, then it can simply update the value of the `tref` in-place.

If it does not own the lock, it tries to acquire it using its thread ID and most significant bit set as the lock value. If it fails to acquire the lock because someone else already owns it, or the compare and swap operation fails, the transaction is aborted. If it successfully acquires the lock, then the `tref` along with the previous value of the `tref` are added to the thread local write set (sometimes referred to as an "undo-log").

5.3 Commit

5.3.1 NOrec Commit

To commit a transaction in NOrec, the thread attempts to increment the global version clock via a compare and swap, such as:

```
CAS(&global_vclock, read_version, MSB | read_version)
```

This allows us to simultaneously check that the thread's read version is still equal to the version clock *and* lock the clock in one atomic instruction. If the compare and swap succeeds, then each entry in the write set is written into the respective `tref`s and finally the version clock is unlocked.

If the compare and swap fails, then the read set needs to be validated as described later in Section 5.4. If validation succeeds, then the compare and swap process is repeated until it eventually succeeds, or an invalid entry in the read set is detected.

5.3.2 TL2 Commit

For TL2, a thread first acquires the lock for every `tref` recorded in the write set using its thread ID and most significant bit set. Since writes are nondestructively consed onto the front of the write set, it is possible that there are duplicate write entries. As we traverse the write set locking `tref`s, if we find one that is already locked by the committing thread, then we know that this is a duplicate

entry that is *older* than the previous entry already encountered. In this case, we simply drop the entry from the write set.

Next, a write version is acquired from the global clock and the read set is validated as described in the next section. If any part of the read set is invalid with respect to the read version, then an abort occurs and returns execution to the point of conflict. If validation succeeds, then for each `tref` in the write set, we write the recorded value into the tref, update the version number associated with the tref to the write version (which simultaneously releases the lock).

5.3.3 TinySTM Commit

The commit protocol is similar to TL2 with the exception that locks do not need to be acquired for `tref`s in the write set (becasue they are acquired inflight). If the read set is valid, then a write version is acquired, and each `tref` in the write set is stamped with this version, unlocking the `tref`. If an entry of the read set is found to be out of date, then execution resumes at the latest safe checkpoint after each `tref` in the write set is reverted. To do this, the write set (undo log) is scanned, replacing the current value with the backed-up value in the log and overwriting the version with the backed up version in the `tref`.

5.4 Read Set Validation

Read set validation is the task of ensuring that each `tref` in the read set is valid with respect to some criteria. This criteria differs between STMs:

- TL2 and TinySTM - `tref` is unlocked and version is less than or equal to read version, or the `tref`'s lock is owned by the validating thread
- NOrec - `tref`'s current value is equal to the value stored in read set entry

During the tail-recursive traversal of the read set, we maintain a checkpoint parameter of type: `('a tref * 'a cont * read_set) option`, where `NONE` corresponds to having seen

no out of date `tref`. If a `tref` is found to be out of date, then we update the checkpoint parameter to `SOME(tr, k, rs)`, where `tr`, `k`, and `rs` are: the `tref` read from, the continuation captured at the read, and the remaining read set respectively. We then traverse the remaining portion of the read set in order to find any earlier conflicts. After traversing the entire read set, if the checkpoint is of the form `SOME(tr, k, rs)`, then we perform the following steps:

- Update the thread's read set to `rs`
- Update the write set to the portion of the write set that `rs` points to (see Figure 5.1)
- Update the read version to the version number received prior to validation (write version)
- Throw to `k`

If, after traversing the entire read set, the checkpoint is of the form `NONE`, then we return a result indicating that validation succeeded.

5.5 Relation to Formal Model

Clearly there are a few differences between the implementation described thus far and the formal semantics presented in Chapter 3. First, the lock is never used for `tref`s in the TL2 semantics, yet it is used in the commit protocol in the actual implementation. The difference here is that in the semantics, the commit is handled in a single atomic step, where as there can be interleaving with other threads in the actual implementation. We believe this to be an implementation detail and does not influence the correctness of partial aborts in an interesting way since the locking protocol is the same for both full-abort and partial-abort.

Second, the metadata for `tref`s in the implementation is slightly different than in the semantics. In the formal model, we backup the old value for TinySTM in the `tref` itself in addition to storing it in the write set. The reason for this is that the READ_REPLAY rule needs to know what the old value of the `tref` is in order to determine what value to continue with, as determined by

	Full Abort	Partial Abort (Unbounded)	Partial Abort (Bounded)
Execution Time	9.220 s	9.271 s	6.836 s
Aborts	11,325	9,150	7,850
GC Time	1.27 s	3.91 s	0.848 s
Allocation	132,549 M	95,401 M	103,898 M

Figure 5.2: TL2 Linked List Stats (Full Abort vs. Partial Abort)

mkVal. If the backed-up value where privately buffered in a thread's write set, it would somehow need access to the lock owner's write set to get that information.

5.6 Garbage Collection

The implementation presented thus far sounds good in theory; however, in practice, it does not yield impressive results. As a preliminary benchmark, we tested the partial abort TL2 implementation on an ordered linked list benchmark, where each thread performs 4,000 operations including lookup, insertion, and deletion from the list. We found that the partial abort implementation performed marginally slower than the full abort reference implementation. When taking a closer look at the performance, we found that keeping a continuation for each `tref` that was read had substantial impacts on garbage collection performance.

Figure 5.2 contains the results of the linked list benchmark. Interestingly enough, the partial abort implementation discussed thus far (Column 2) aborts fewer transactions, causing it to perform less work, and in turn allocate less data, yet spends 3X time performing garbage collection compared to full abort. The reason for these unexpected results is due to the liveness of the continuations being recorded in the read set. In the full abort implementation, a return continuation is allocated, passed into the read function, the `tref` is read from, and the return continuation is thrown to. Once the return continuation is invoked, there remain no more references to it, allowing the garbage collector to reclaim the space taken up by the closure. In the partial abort implementation, however, we maintain a pointer to this closure until either an abort takes place or the

transaction commits. For the linked list benchmark, the read sets become very large (4,000+ entries), causing a substantial discrepancy in the heaps between the full abort and partial abort (with an unbounded number of continuations) implementations.

5.7 Bounding Continuations

In an effort to deal with the garbage collection issue, we have devised a scheme to limit the number of continuations held by any transaction to a constant factor. This constant factor is determined based on the size of the heap, rather than tuned in an application-specific manner. The same constant is used for each benchmark presented in Chapter 6 for each STM implementation.

The first change made to support bounded continuations is that elements in the read set may or may not contain a continuation. This requires a slight modification of the commit and eager detection code, where we now revert control to the latest safe checkpoint, which is not necessarily the exact point of the conflict. Second, we have changed the representation of the read set from a traditional linked list to a skip list as depicted in Figure 5.3. There still exists a long path, which passes through every node in the read set; however, there is also now a short path which only passes through items in the read set that contain a continuation. Lastly, each thread maintains a counter that controls the frequency at which continuations are captured.

Threads begin by capturing a continuation at every read from a `tref`. As soon as the maximum number of continuations is reached (20 in our implementation), we walk the short path of the read set and drop the continuation for every other entry. Then the frequency is updated to capture a continuation at every other read. Figure 5.3 shows the read set after this filtering has occurred, so when the next `tref` is read from (T_6), we will not capture a continuation and will not add it to the short path, but when T_7 is read, a continuation will be captured and added to the short path. Once the maximum is reached a second time, we again drop every other continuation and start capturing every 4 continuations. The frequency continues to double each time the bound is reached and the read set is filtered.

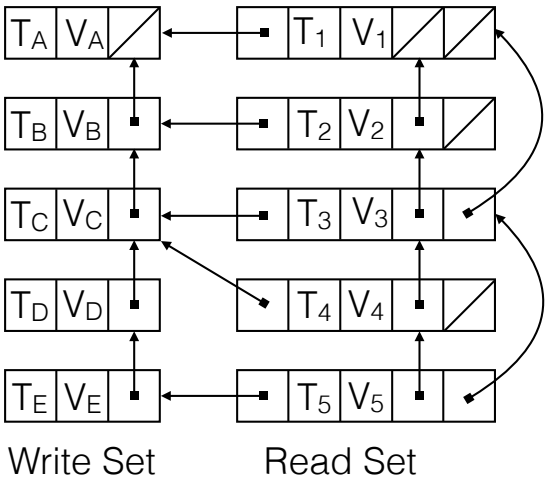

Figure 5.3: Skip List Representation of Read Set

This approach allows us to limit the number of continuations to a constant factor, while maintaining an even distribution of checkpoints throughout the transaction: even if a conflicting read does not have a continuation, the latest safe checkpoint will nonetheless salvage a good portion of the transaction. It is also worth noting that by using the skip list, we can perform the filtering operation in constant time.

Looking back at Figure 5.2, we can see that this does in fact have dramatic savings in just about every respect. The execution time improved by nearly 26% relative to the full abort implementation. Additionally, the number of aborted transactions was reduced even further compared to the partial abort implementation with unbounded continuations. The amount of allocated data sits somewhere between full and unbounded partial abort. It is less than full abort implementation, because fewer transactions are being aborted, so less work is being done, corresponding to less allocation. However, the read set requires that additional information be maintained and uses slightly more space than the unbounded partial abort implementation. That said, the time spent doing GC is substantially better than unbounded partial abort and slightly better than full abort. The improved garbage collection time over full abort can be attributed to the fact that less data is being allocated.

One interesting aspect to this design is that we are using mutation to filter elements from the

read set, yet we do not violate any of the generational garbage collector invariants. Typically when mutation is involved, the mutated heap object needs to be added to some sort of remember set so that it can be treated as part of the root set in case the object it is being updated to point to is in a younger generation. In the case of our skip list representation, the only types of mutations that we perform are setting fields to NULL and updating a short path pointer to point to an object that is *deeper* in the structure than what it is already pointing at. Clearly setting something to NULL cannot violate any invariants, but the reasoning for the latter scenario is a bit more subtle.

We can informally prove that the generational invariant is not violated by induction. Assume that the short path pointer is currently pointing to an item in the read set that is as old or older than itself and that all items closer to the head of the list are newer than items further down the list. If the read set needs to be filtered, then we will either set the next pointer to NULL, or it will be pointing to an item deeper in the list than it is already pointing at. By our induction hypothesis this is older than the object we were already pointing at, and is thus older than the heap object we are updating, which preserves the generational invariant.

5.8 Chronologically Ordered Read Sets

One downside to the representation we have chosen for our read set is that the entire list needs to be scanned to detect a conflict when performing partial aborts. Since a read item is consed onto the head of the list each time, the natural order of traversing the list corresponds to the reverse chronological order. This is fine for the full abort implementation, since it is only concerned with whether a conflict exists or not; thus, when traversing in reverse chronological order, as soon as a conflict is found the transaction can abort without looking at the rest of the list. When performing partial aborts, in order to preserve correctness, we must scan the entire list, to ensure that the chronologically earliest conflict is found.

As mentioned in Section 4.2, the split heap representation used in Manticore requires that we maintain two invariants. First, there cannot be any pointers from the global heap into any

local heap, and second, there cannot be any pointers from a local heap into another local heap. Implementing a chronologically ordered read set can potentially violate the first invariant. If a garbage collection occurs in a local heap, it is possible that the read set can get promoted to the global heap. If we then allocate a new node for an entry in the read set and append it on to the end of the list, we will have the tail of the linked list, which exists in the global heap, pointing to a newly allocated node that exists in a local heap. The naïve way to get around this is to promote newly allocated elements into the global heap before appending them onto the end of the list. Unfortunately, in order to preserve the heap invariants, everything transitively reachable also needs to be promoted, which includes the closure of the return continuation, adding significant overhead to every read.

In order to get around this, we have extended Manticore with a remember set for heap objects that violate the previously mentioned heap invariants. Each time we perform a read, we check to see if the last entry in the read set is in the local heap, which is done by comparing the address to the beginning and end addresses of the local heap. If it is in the local heap, then we simply append the new entry via mutation. If the last entry happens to have already been promoted, we add the address and offset we are updating to the remember set (stored in the VProc). When the next garbage collection happens, the remember set is scanned and treated as part of the root set.

CHAPTER 6

EVALUATION

Our benchmark machine is a Dell PowerEdge R815 machine, equipped with 48 cores and 128 GB of physical memory. This machine runs x86_64 Ubuntu Linux 16.04.1 LTS, kernel version 4.4.0-31. The 48 cores are provided by four 12 core AMD Opteron 6172 "Magny Cours" processors; each core operates at 2.1 GHz and is equipped with 64 KB of instruction and data L1 cache and 512 KB of L2 cache; each processor is equipped with two 6 MB L3 caches (each of which is shared by six cores).

6.1 Benchmarks

To quantify the performance of partial aborts, we have selected a number of benchmarks from the STAMP benchmark suite [8] as well as a few common data structures used to evaluate transactional memory implementations. The evaluation covers TL2, TinySTM, NOrec, and their partial abort analogs, each implementation makes use of the bounded continuation optimization (limited to 20 checkpoints) described earlier and uses a chronologically ordered read set. Benchmarks were chosen to provide a wide spectrum of workloads including long transactions, short transactions, and a mix of the two.

Linked List Linked List implements an ordered linked list, where each node in the linked list is represented as a `tref`. The list is (sequentially) initialized with 500 elements and then each thread begins inserting, deleting, and querying the list with a ratio of 1:1:1. The benchmark consists of long transactions, which presents good opportunities for partial aborts.

Labyrinth (STAMP) Labyrinth implements Lee's parallel routing algorithm [44]. The objective is to find a path for all source-destination pairs concurrently without having any overlapping paths. This benchmark exhibits very long transactions with large write sets.

Red Black Tree Red Black Tree implements a concurrent self balancing binary search tree, where each node is protected by a `tref`. The tree is (sequentially) initialized with 100,000 elements and then each thread performs 500,000 operations, consisting of queries, insertions, and deletions with a ratio of 1:1:1. This benchmark exhibits medium-length transactions.

Skip List Skip list implements a skip list data structure that defaults to 14 levels. The skip list is initialized with 100,000 entries and then each thread performs insertions, lookups, and deletions with a ratio of 1:1:1.

Vacation (STAMP) Vacation simulates a travel reservation system. The reservation system consists of a database represented as a binary search tree with a `tref` at each node. Clients are able to make and cancel reservations and the travel reservation system is able to add and remove available reservations. This benchmark exhibits medium length transactions.

KMeans (STAMP) KMeans implements a clustering algorithm commonly used in data mining and machine learning. A transaction is used to protect the update of the cluster centers, which amounts to incrementing a counter by a constant. This benchmark consists of very small transactions (1 read and 1 write) permitting zero opportunities for partially-aborting a transaction.

6.2 Benchmark Results

Figure 6.1 presents results for the previously described benchmarks. Each benchmark is run on range of 1 - 48 cores, 5 times for each core, which is then averaged for each data point. It is important to make note of the y-axis for each plot as Linked List, Red Black Tree, and Skip List are all presented in terms of throughput (higher is better), whereas KMeans, Labyrinth, and Vacation are reported in terms of execution time (lower is better). It is necessary to measure the data structure benchmarks in terms of throughput because there can be a wide variation in terms of performance between various STMs, as can be seen in Skip List and Red Black Tree. If we

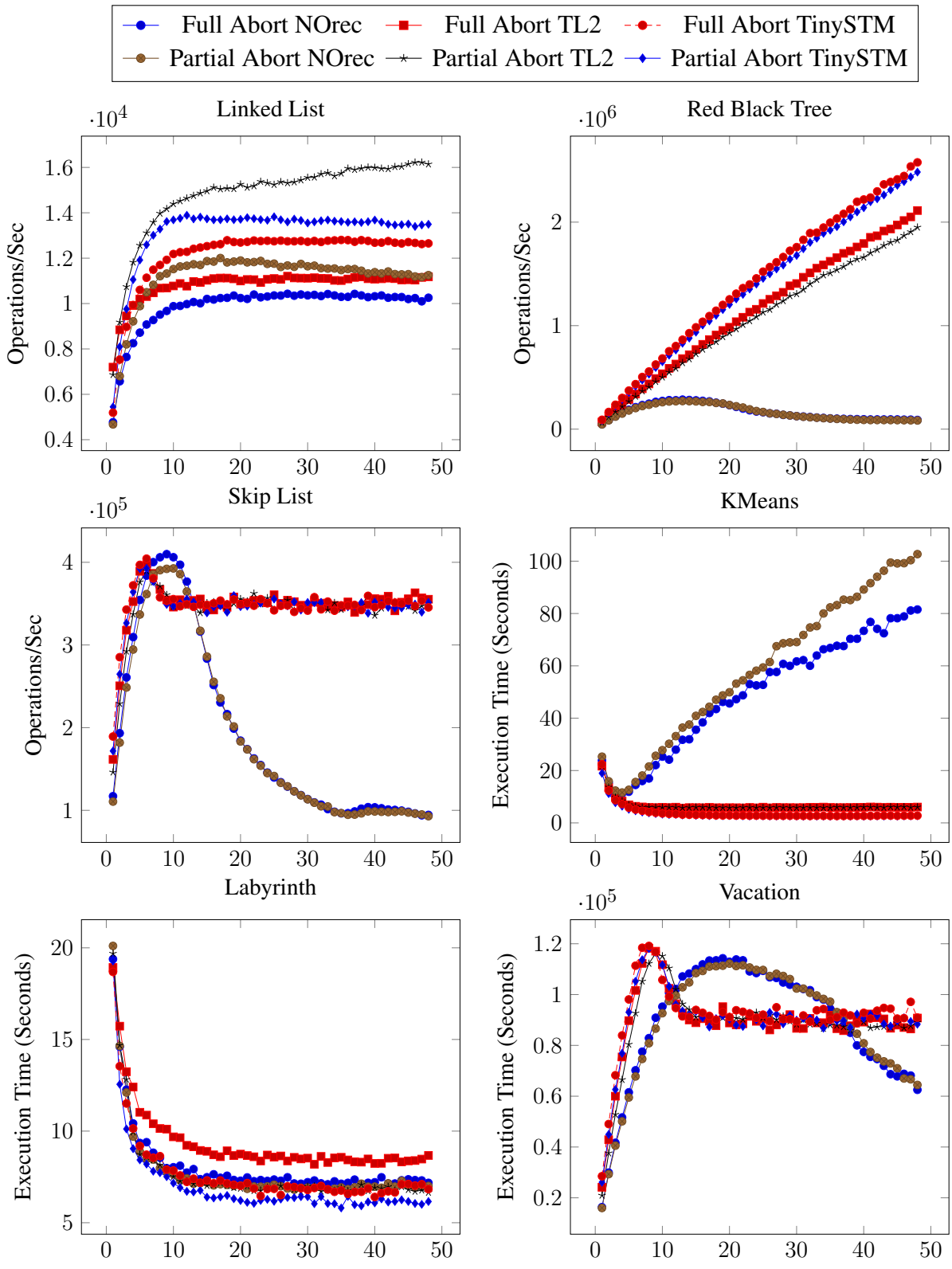

Figure 6.1: Benchmark Results

were instead going to settle on a fixed work load (i.e. number of operations to be performed) and measure the execution time, NOrec would take an unreasonable amount of time to finish, or TL2/TinySTM would not be doing a sufficient amount of work to get an accurate measure of execution time.

As is expected, the benchmarks that contain many large transactions benefit the most from performing partial aborts. All three STM implementations perform quite well on the linked list benchmark, with TL2 achieving the most substantial improvement. The reason for this is the lack of a timestamp extension mechanism in the STM. When a conflict is detected eagerly (mid-transaction), TL2 aborts the transaction. For the partial-abort implementation, we must traverse the read set in order to figure out where to abort to. It is often the case that the abort is able to salvage a substantial portion of the transaction. Additionally, we believe that the sequential bottleneck in the NOrec commit protocol inhibits scalability which also dampens the separation between full-abort and partial-abort. The partial abort implementations also do quite well on the Labyrinth benchmark, with TL2 and TinySTM having the largest margins compared to their full-abort counterparts.

The two cases where partial aborts end up not doing well is in the KMeans (NOrec) and Red Black Tree (TL2 and TinySTM) benchmarks. For KMeans, both NOrec implementations perform quite poorly. In the KMeans benchmark, transactions are used only to protect the cluster centers, where each thread simply performs a bunch of individual fetch and adds. This leads to a huge number of very short writing transactions, which is exactly NOrec's Achille's heel.

Red Black Tree also shows a bit of a deviation where partial-abort is falling behind. After some further investigation, in turns out that many of the aborts that occur take place early on in the transaction. The problem is that when inserting or deleting from the tree, a path of nodes is read until the desired node is found. After inserting or deleting, the thread then rebalances the tree, which ends up re-reading that same path of nodes. If a conflict occurs on any node on that path, then we must abort back to the first read from that `tref`. In this case, the partial abort ends up not

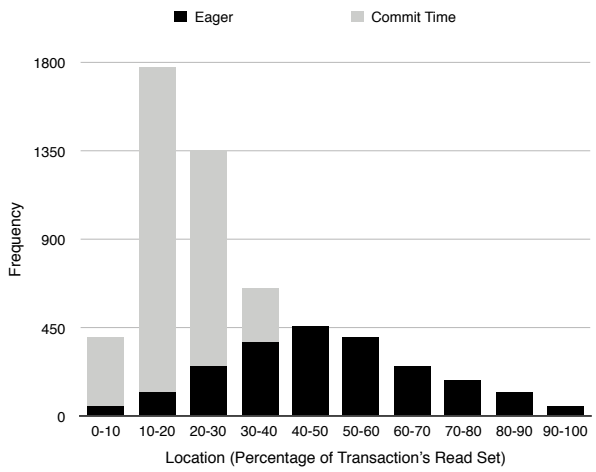

Figure 6.2: Red Black Tree Partial Abort Positions

providing much benefit.

Figure 6.2 contains a histogram describing this phenomenon. The bars are split into two categories, the dark colored portion represents conflicts that were detected eagerly (during a read) and the light colored portion represents conflicts that were detected at the end of a transaction. The x-axis indicates what portion of the transaction the thread partially aborted to with respect to the length of the read set. We can see that the vast majority of the aborts restored control to a position within the first 30% of the transaction. Note that almost all conflicts that aborted to the 30-100% portion of the transaction were eager conflicts, so the total number of reads performed at the point of validation is less, yielding a smaller benefit than a partial abort performed at commit time.

The remaining benchmarks, Skip List and Vacation, do not show much of a difference between the respective partial-abort and full-abort implementations. The main takeaway from this evaluation is that partial aborts in general do not do damage in terms of performance, but in applications with long running transactions it can provide substantial improvements.

Hardware transactional memory has recently seen a lot of attention in the transactional memory research community where the cache coherence protocol is used to determine conflicts. The glaring issue with hardware transactional memory is that it is considered "best effort," meaning if the cache overflows due to limitations in the size or associativity of the cache, the transaction also

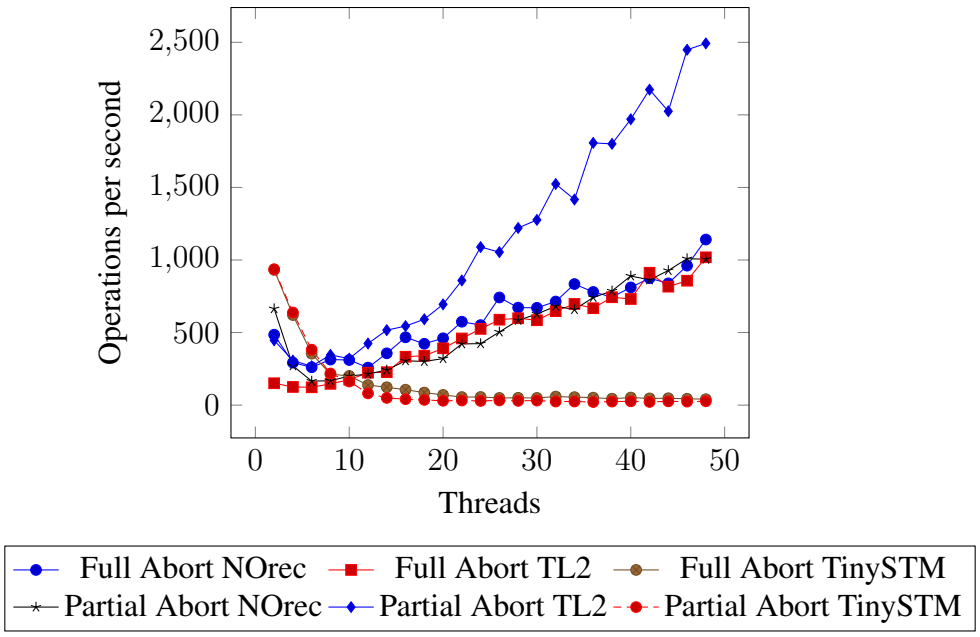

Figure 6.3: Half and Half Linked List Benchmark

fails. In order to deal with this problem, hybrid approaches [10] have been built, such that if a hardware transaction fails too many times, it falls back to a software based transactional memory. We believe that an STM that is able to provide partial aborts would be an excellent complement to a hardware based TM. Short running transactions do not benefit much from partial aborts, but are what hardware TM systems handle very well. On the other hand, long running transactions are the bane of hardware TM systems, and where our partial abort STM really shines.

6.3 Throughput

One of the arguments we use to motivate partial aborts is that of fairness and throughput. In a context where some threads are executing short transactions that conflict with long running transactions, we would prefer the probability of a transaction committing to be as uniform as possible. To that effect, we have evaluated the throughput of partial aborts on an ordered linked list benchmark, where half of the threads perform their operations only on the first 50% of the linked list and the other half perform their operations only on the second half of the list. Thus, the threads

operating in the first half have a much higher probability of committing their transaction. Each execution is run for 5 seconds and the total number of operations per second completed by the threads working in the second half of the list is recorded.

Figure 6.3 contains the results of this experiment, giving the number of completed operations performed by second-half threads. Operations are only counted for the threads working in the second half of the list, as they are the ones at a disadvantage that we are interested in quantifying. Note that the number of threads along the x-axis indicates the total number of threads in the benchmark, so at the 48 core mark, there are 24 threads operating in the first half of the list and 24 threads working in the second half of the list.

We find that the partial-abort TL2 implementation substantially outperforms its full-abort counterpart by nearly a factor of two when scaling all the way out to 48 cores. TinySTM and NOrec on the other hand do not show much deviation from one another. TinySTM in particular performs very poorly on this benchmark (both partial-abort and full-abort). We believe this to be caused by the encounter time locking that TinySTM does when writing to a `tref`. When a transaction encounters a locked `tref` mid-transaction it polls the `tref` waiting for a bounded amount of time in the hopes that it will soon be released and it can continue with the transaction. As a result, when a `tref` is locked, other threads end up waiting mid point in their transactions for the lock to be released. When the lock is eventually released, thread's resume their transactions without having to restart from the beginning. With TL2 on the other hand, threads must restart their transactions, leaving more time for the second half threads to reach their commit point. The lack of difference in the NOrec implementation can be attributed to the scalability bottleneck inherent in the commit protocol.

CHAPTER 7

RELATED WORK

The most closely related work to what is presented in this thesis is that of Koskinen and Herlihy [26], where the authors first proposed partially aborting transactions. The main difference is in the implementation of partially aborting transactions. In that work, the authors need to perform stack copying in order to safely revert control to a checkpoint in the event of a violation. In this work, we make use of the CPS transformation to perform checkpointing much more efficiently. Additionally, we provide a novel mechanism for controlling the number of checkpoints created that allows us to transparently perform the checkpointing. It is worth noting that our checkpoint bounding mechanism is unique to our situation. The issue encountered by Koskinen and Herlihy [26] is that creating checkpoints is very expensive. In our case, creating checkpoints is cheap, but keeping them live incurs overhead. The bounding approach critically relies on the fact that it can control the frequency of checkpoints based on the transaction it has seen so far, rather than trying to predict what the size of the transaction is going to be in the future.

Gupta et al. also explored checkpointing transactions [22]. They attempt to control the number of checkpoints created by associating a conflict probability with each transactional location based on the number of times it is accessed within a transaction. Additionally, they use a frequency counter similar to what we present, however, this is a uniform constant that does not adapt as the transaction proceeds. This constant, is then application specific and would need to be tuned for each program. They also do not have an actual implementation and instead report results based on a simulator. This approach makes it difficult to address the true overheads associated with creating checkpoints, which as reported by Koskinen and Herlihy [26] is substantial when not performing a CPS conversion.

Nested transactions have been proposed in a number of variations [30, 31, 23], where atomic blocks can be nested arbitrarily. At the end of an atomic block, the read set is validated, and the transaction commits after the outermost atomic block can be validated. Figure 7.1 shows how

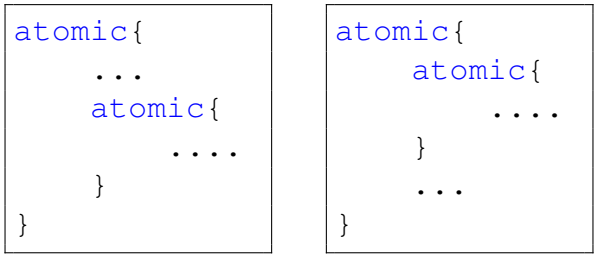

Figure 7.1: Common Nested Transactional Idioms

nested transactions are commonly used as a checkpointing mechanism. On the left, we have a nested transaction towards the end of the outermost transaction. If validation fails in the inner transaction, then execution returns to the beginning of the second atomic, rather than going all the way back to the beginning. In the second example, the inner atomic is placed at the beginning, so that the log will be validated early in the hope of not wasting time executing the remainder of the outermost transaction if there is a violation at the beginning. These two idioms are essentially subsumed by checkpointing and eager conflict detection.

Timestamp extension [39] has been used in many recently proposed STM systems [16, 42, 38], where a new stamp is fetched from the global clock and the read set is validated when an eager conflict is detected. If the entire read set can be validated, then the transaction is able to proceed with the new time stamp, avoiding many spurious aborts. Unfortunately timestamp extension is an all-or-nothing operation, meaning that if a thread attempts a timestamp extension and it fails, then the entire timestamp extension operation is wasted effort. As was mentioned in Section 5.1, eager conflict detection with partial aborts generalizes this technique by additionally being able to salvage a portion of the transaction if validation of the entire log fails, yielding value even in the presence of failure.

Ziarek et al. proposed a language abstraction called the *stabilizer* [48], which establishes a checkpoint in the context of concurrent message passing and transient faults. If a thread needs to re-execute a section of code due to a transient fault that includes message passing communication with another thread, then all threads involved revert to a safe checkpoint. This requires that tran-

sitive dependencies be tracked via an incremental graph construction scheme. Additionally, since checkpoints are created manually, they do not encounter the same problems that lead us to our bounded continuation optimization.

Checkpointing is a fundamental part of recent work on self-adjusting computation [28]. In that work, a selective CPS transformation is used for functions that are annotated as self adjusting so that continuations can efficiently be captured. The authors note significant overheads due to maintaining pointers to continuations and report all times without time spent doing garbage collection. We believe that our approach to bounding the number of continuations held at any given point could be used to solve this problem.

CHAPTER 8

CONCLUSION

The work in this thesis presented an extension of a full-abort transactional memory algorithm that is able to efficiently support partial aborts for transactions. Previous attempts at this have required that checkpoints be explicitly inserted by the programmer, which we argue is burdensome and ineffective. Many of the benchmarks presented in Chapter 6 have unpredictable abort patterns. For example, with the linked list benchmark, there is equal probability of aborting at every `tref` in the linked list read from, which does not lead to any obvious point to manually place a checkpoint. Our approach to bounding the number of continuations maintained in the read set automatically learns the right granularity to capture continuations, leading to an efficient and easy to use implementation.

A second argument for transparently checkpointing transactions, is that it adapts with the composition of transactions. Compositionality is one commonly cited attractive feature of STM. If programmers are to manually insert checkpoints in their code, it is possible that a checkpoint makes sense in a given context, but when composed with other transactions, no longer has desirable performance. By automatically adjusting the frequency at which continuations are captured on a per transaction basis, we are able to find the right granularity regardless of composition.

As noted in Chapter 6, we believe that our partial abort STM would make an excellent software fallback for hybrid transactional memory systems. Hardware transactions would be able to take care of the short running transactions, which our partial abort system is not able to do much with. On the other hand, the long running transactions that hardware is unable to support would be handled very efficiently by our partial abort STM.

We credit the initial design decisions of the Manticore runtime system for the elegance and simplicity of our implementation. Basing the scheduling infrastructure on first-class continuations led to very flexible scheduling policies [32], but also allowed us to implement our partial abort STM extension quite easily. The entire implementation is less than 400 lines of BOM code and

did not require any modifications of the compiler or core runtime system. Furthermore, we believe that a number of extensions to our STM library can easily be added on top with little effort. For example, we could easily add manual checkpointing in the following manner:

```
local val cpTRef = STM.new 0
in fun checkpoint() = (STM.get cpTRef; ())
end
```

Since no thread has the ability to write to `cpTRef`, it will always serve as a safe checkpoint in a thread's read set.

REFERENCES

[1] A. W. Appel. Simple generational garbage collection and fast allocation. 19(2):171–183, 1989.

[2] A. W. Appel. *Compiling with Continuations*. 1992.

[3] S. Auhagen, L. Bergstrom, M. Fluet, and J. Reppy. Garbage Collection for Multicore NUMA Machines. 2011.

[4] A. Baldassin, E. Borin, and G. Araujo. Performance implications of dynamic memory allocators on transactional memory systems. pages 87–96, 2015.

[5] L. Bergstrom, M. Fluet, M. Le, J. Reppy, and N. Sandler. Practical and effective higher-order optimizations. pages 81–93, 2014.

[6] L. Bergstrom and J. Reppy. Arity raising in manticore. pages 90–106, 2009.

[7] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. pages 99–107, 1996.

[8] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08*, September 2008.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. In *PPoPP '10*, pages 67–78, New York, NY, USA, 2010. ACM.

[10] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, Oct. 2006.

[11] O. Danvy and A. Filinski. Representing control: A study of the CPS transformation. 2(4):361–391, 1992.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. volume 4167, pages 194–208, 2006.

[13] N. Diegues and P. Romano. Time-warp: Lightweight abort minimization in transactional memory. pages 167–178, 2014.

[14] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. pages 70–83, Jan.

[15] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. pages 113–123, Jan. 1993.

[16] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. pages 237–246, 2008.

[17] K. Fisher and J. Reppy. Compiler support for lightweight concurrency. Technical memorandum, Bell Labs, Mar. 2002. Available from `http://moby.cs.uchicago.edu/`.

[18] M. Fluet, N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. pages 15–24, 2007.

[19] M. Fluet, M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. pages 241–252, 2008.

[20] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. 20(5–6):537–576, 2011.

[21] L. George, F. Guillame, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. pages 83–97.

[22] M. Gupta, R. K. Shyamasundar, and S. Agarwal. Clustered checkpointing and partial rollbacks for reducing conflict costs in stms. *International Journal of Computer Applications*, 1(22):82–87, 2010.

[23] T. Harris and S. Stipic. Abstract nested transactions. In *TRANSACT 2007*, January 2007.

[24] C. T. Haynes, D. P. Friedman, and M. Wand. Continuations and coroutines. pages 293–298, 1984.

[25] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*, pages 289–300, 1993.

[26] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA '08*, pages 160–168, 2008.

[27] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. pages 19–30.

[28] R. Ley-Wild, M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. pages 321–334, 2008.

[29] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. 1997.

[30] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS XII*, pages 359–370, New York, NY, USA, 2006. ACM.

[31] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP '07*, pages 68–78, New York, NY, USA, 2007. ACM.

[32] M. Rainey. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. PhD thesis, University of Chicago, Aug. 2010. Available from `http://manticore.cs.uchicago.edu`.

[33] N. Ramsey. Concurrent programming in ML. Technical Report CS-TR-262-90, Apr. 1990.

[34] N. Ramsey and S. Peyton Jones. Featherweight concurrency in a portable assembly language. Unpublished paper available at `https://www.cs.tufts.edu/~nr/pubs/c--con-abstract.html`, Nov. 2000.

[35] J. Reppy, C. Russo, and Y. Xiao. Parallel Concurrent ML. pages 257–268, 2009.

[36] J. H. Reppy. First-class synchronous operations in Standard ML. Technical Report TR 89-1068, Dec. 1989.

[37] J. H. Reppy. *Concurrent Programming in ML*. 1999.

[38] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. volume 4167, pages 284–298, 2006.

[39] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07*, pages 221–228, New York, NY, USA, 2007. ACM.

[40] N. Shavit and D. Touitou. Software transactional memory. pages 204–213, 1995.

[41] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. 1997.

[42] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09*, pages 141–150, New York, NY, USA, 2009. ACM.

[43] M. Wand. Continuation-based multiprocessing. pages 19–28, 1980.

[44] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT '07*, pages 388–398, Washington, DC, USA, 2007. IEEE Computer Society.

[45] L. Xiang and M. L. Scott. Composable partitioned transactions. In *WTTM'13*, 2013.

[46] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *PPoPP 2015*, pages 76–86, New York, NY, USA, 2015. ACM.

[47] M. Zhang, J. Huang, M. Cao, and M. D. Bond. Low-overhead software transactional memory with progress guarantees and strong semantics. pages 97–108, 2015.

[48] L. Ziarek, P. Schatz, and S. Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. pages 136–147, 2006.

APPENDIX A

TL2 AND NOREC LOG VALIDATION RULES

$$\boxed{\textbf{validate}_{\text{norec}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(L', e'; H_i); H_v) \mid \textbf{abort}(L'; e'; H_i)}$$

$$\frac{S = C}{\textbf{validate}_{\text{norec}}(id; e_0; S; \cdot; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(\cdot; e_0; H); H)} \text{ NORECEMPTY}$$

$$\frac{S = C \quad \textbf{validate}_{\text{norec}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H)}{\textbf{validate}_{\text{norec}}(id; e_0; S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H)} \text{ NORECWPROPABORT}$$

$$\frac{S = C \quad \textbf{validate}_{\text{norec}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H)}{\textbf{validate}_{\text{norec}}(id; e_0; S; L, \ell \mapsto_c (\mathcal{E}, v); H; C) \rightsquigarrow \textbf{abort}(L'; e'; H)} \text{ NORECRPROPABORT}$$

$$\frac{S = C \quad \textbf{validate}_{\text{norec}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v)}{\textbf{validate}_{\text{norec}}(id; e_0; S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v, \ell \mapsto (v, (S)_u))} \text{ NORECVWRITE}$$

$$\frac{S = C \quad \textbf{validate}_{\text{norec}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v, lock)}{\textbf{validate}_{\text{norec}}(id; e_0; S; L, \ell \mapsto_c (\mathcal{E}, v); H; C) \rightsquigarrow \textbf{commit}(L; \mathcal{E}[!\ell]; H; H_v)} \text{ NORECVREAD}$$

$$\frac{S = C \quad \textbf{validate}_{\text{norec}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v', lock) \quad v \neq v'}{\textbf{validate}_{\text{norec}}(id; e_0; S; L, \ell \mapsto_c (\mathcal{E}, v); H; C) \rightsquigarrow \textbf{abort}(L; \mathcal{E}[!\ell]; H)} \text{ NORECAREAD}$$

Figure A.1: NOrec Transactional Log Validation

$$\boxed{\textbf{validate}_{\text{tl2}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(L', e'; H_i); H_v) \mid \textbf{abort}(L'; e'; H_i)}$$

$$\frac{}{\textbf{validate}_{\text{tl2}}(id; e_0; S; \cdot; H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(\cdot; e_0; H); H)}\ \text{TL2EMPTY}$$

$$\frac{\textbf{validate}_{\text{norec}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H)}{\textbf{validate}_{\text{tl2}}(id; e_0; S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H)}\ \text{TL2WPROPABORT}$$

$$\frac{\textbf{validate}_{\text{norec}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{abort}(L'; e'; H)}{\textbf{validate}_{\text{tl2}}(id; e_0; S; L, \ell \mapsto_c (\mathcal{E}, v); H; C) \rightsquigarrow \textbf{abort}(L'; e'; H)}\ \text{TL2RPROPABORT}$$

$$\frac{\textbf{validate}_{\text{tl2}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v)}{\textbf{validate}_{\text{tl2}}(id; e_0; S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v, \ell \mapsto (v, (C)_u))}\ \text{TL2VWRITE}$$

$$\frac{\textbf{validate}_{\text{tl2}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v, (S)_u) \quad S < S'}{\textbf{validate}_{\text{tl2}}(id; e_0; S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \text{chkpnt}}\ \text{TL2AWRITE}$$

$$\frac{\textbf{validate}_{\text{tl2}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v, (S')_u) \quad S \geq S'}{\textbf{validate}_{\text{tl2}}(id; e_0; S; L, \ell \mapsto_c (\mathcal{E}, v); H; C) \rightsquigarrow \textbf{commit}(\textbf{abort}(L; \mathcal{E}[!\ell]; H); H_v)}\ \text{TL2VREAD}$$

$$\frac{\textbf{validate}_{\text{tl2}}(id; e_0; S; L; H; C) \rightsquigarrow \textbf{commit}(\text{chkpnt}; H_v) \quad H(\ell) = (v, (S)_u) \quad S < S'}{\textbf{validate}_{\text{tl2}}(id; e_0; S; L, \ell \mapsto_c (\mathcal{E}, v); H; C) \rightsquigarrow \textbf{abort}(L; \mathcal{E}[!\ell]; H)}\ \text{TL2AREAD}$$

Figure A.2: TL2 Transactional Log Validation