Partial Aborts for Transactions

Matthew Le

Rochester Institute of Technology, USA
ml9951@cs.rit.edu

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.

In this paper, we present 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. We have implemented this idea

in the context of the Manticore project, an ML-family language with support for parallelism and concurrency. Crucially, we rely on constant-time continuation capturing via a continuation-passing-style (CPS) transformation and heap-allocated continuations. When comparing our STM that performs partial aborts against one that performs full aborts, we achieve a decrease in execution time of up to 31% and an increase in throughput of up to 351%.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors Run-time environments

General Terms Languages

Keywords Software Transactional Memory, First-Class Continuations

1. Introduction

Software transactional memory (STM) [ST95, HM93] 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 syn-

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08
http://dx.doi.org/10.1145/2784731.2784736

via First-Class Continuations

Matthew Fluet
Rochester Institute of Technology, USA
mtf@cs.rit.edu

chronization 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. In this work, we build on an algorithm that uses lazy versioning, meaning that updates to shared references are not visible to other threads until the end of the transaction. In this scheme, the runtime system maintains a thread-local log recording which references were read from and written to within a transaction. When a thread writes to a reference, rather than modifying memory directly, it creates a local copy of the reference and records the written value on the copy. At the end of the transaction, the thread validates its log and if no conflicts are detected, it commits all of the local copies

to the global store. If a conflict is detected, then it throws away the log 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 [SDMS09], 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 [Wan80, Shi97, RRX09] and serves as a fundamental component of the Manticore scheduling infrastructure [FRR08]. In this work we make 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 [KH08]. 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 man-

ually 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 paper makes the following contributions:

- We present an extension of the Transactional Locking II algorithm, a modern, high performance STM algorithm, to partially-abort transactions.
- We identify a significant overhead in garbage collection due to live captured continuations and present a scheme to bound the

number of continuations held to a constant factor.

- We formalize the semantics of STM that performs partial aborts and give a machine checked proof, using the Coq proof assistant, that it yields equivalent final program states to a similar implementation that performs full aborts.
- We present a detailed evaluation 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.

2. Baseline STM

We begin by describing the baseline full abort reference implementation that we later extend in Section 5. The full abort algorithm that we compare against is based on the Transactional Locking II (TL2) algorithm [DSS06]. TL2 is one of the top performing implementations of STM and is commonly used in evaluating new STM algorithms [DR14, BBA15, ZHCB15]. The main novelty of TL2 is its use of a global version clock for eagerly detecting conflicts and ensuring atomicity. In this system, threads perform an atomic increment of 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 references that have been altered since the start of the transaction. Additionally, each reference has a version number and a lock; the version number indicates when the reference was last updated.

When a thread writes to a reference, it performs its write on a thread local copy that it maintains in its write set. When reading

from a reference, the thread first consults its write set to check if it has already made updates to the reference. If so, it reads the value of its most recent update to the reference from its write set. If no local copy exists, then it checks that the version number associated with the reference is older than the read version it received at the beginning of the transaction and that the reference's lock is not held. If these checks succeed, then it records the fact that it read from the reference in its read set. If the version number associated with the reference is newer than the read version or the reference's lock is held, then the log is discarded and the transaction is aborted and restarted.

When committing a transaction, the thread first acquires the locks associated with each reference 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 reference read, the version number associated with the reference 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

Values	v	::=	$\lambda x.e \mid \ell \mid ()$
Expressions	e	::=	$v \mid x \mid e\ e \mid \textbf{spawn}\ e$
		$\mid$	$!e \mid e := e \mid \textbf{tref}\ e$
		$\mid$	$\textbf{atomic}\ e \mid \textbf{inatomic}(e)$
Evaluation Context	$\mathcal{E}$	::=	$[\cdot] \mid \mathcal{E}\ e \mid v\ \mathcal{E}$
		$\mid$	$!\mathcal{E} \mid \mathcal{E} := e \mid v := \mathcal{E} \mid \textbf{tref}\ \mathcal{E}$
		$\mid$	$\textbf{inatomic}(\mathcal{E})$
Heap	H	::=	$\cdot \mid H, \ell \mapsto (v, S)$
Thread Pool	T	::=	$\cdot \mid \langle t; e\rangle \mid T \cup T$
Transaction Info	t	::=	$\cdot \mid \langle S; L; e\rangle$
Log	L	::=	$\cdot \mid L, \ell \mapsto_w v \mid L, \ell \mapsto_r \mathcal{E}$

Version Numbers $S, C \quad ::= \quad \mathbb{N}$

Figure 1. Syntax

set is successfully validated, then an atomic increment of the global version clock is performed to retrieve a new version number that 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 reference, the write version is written into the version number associated with the reference, and the lock is released.

This approach has received much praise for its ability to provide a strong guarantee known as opacity [GK08] at a very low performance cost. Opacity is a property that was proposed for STM that requires three conditions hold:

1. For each committed transaction, all operations must appear to the rest of the system as if they were performed as one atomic operation.
2. Threads can not observe any operation performed by an aborted transaction.
3. Every transaction must always maintain a consistent view of memory.

As an example of opacity at work, consider the following program:

```
val get : 'a STM.tref -> 'a = STM.get
val atomic : (unit -> 'a) -> 'a = STM.atomic
fun trans() =
```

```
  let val x = get tref1
      val y = get tref1
  in if x = y then () else infiniteLoop()
  end
val _ = atomic trans
```

In this example, `atomic` takes a function of type `unit -> 'a` that is run atomically and `get` returns the value of a `tref`. As mentioned previously, every time a read is performed, the thread checks that the version associated with the `tref` is older than the version number it received at the start of its transaction. If not, then the transaction is aborted. If this check were not performed, then it is possible that in the above example, `tref1` is modified in between the two reads, changing its value, and causing this to go into an infinite loop. However, with eager conflict detection, a conflict would be detected at the second read, and the transaction would be aborted. By enforcing opacity, users are given a much more intuitive notion of atomicity with which to work.

3. Semantics

We extend the baseline full abort algorithm with the ability to partially abort transactions by resuming execution at the point of a conflict, rather than always resuming execution at the beginning of the transaction. We first present a formal semantics of our exten-

$$\boxed{C; H; T \to_x C'; H'; T' \qquad x \in \{\text{full}, \text{partial}, \text{replay}\}}$$

$$\frac{C; H; T_1 \to_x C'; H'; T_1'}{C; H; T_1 \cup T_2 \to_x C'; H'; T_1' \cup T_2} \text{ PARL} \qquad \frac{C; H; T_2 \to_x C'; H'; T_2'}{C; H; T_1 \cup T_2 \to_x C'; H'; T_1 \cup T_2'} \text{ PARR}$$

$$\frac{}{C;H;\langle\cdot;\mathcal{E}[\mathbf{spawn}\ e]\rangle \to_x C;H;\langle\cdot;\mathcal{E}[()]\rangle \cup \langle\cdot;e\rangle}\ \text{SPAWN} \qquad \frac{}{C;H;\langle t;\mathcal{E}[(\lambda x.e)\ v]\rangle \to_x C;H;\langle t;\mathcal{E}[e[x \mapsto v]]\rangle}\ \text{BETA}$$

$$\frac{\ell \notin \mathrm{Dom}(L|_w) \quad H(\ell) = (v, S') \quad S' < S}{C;H;\langle\langle S;L;e_0\rangle;\mathcal{E}[!\ell]\rangle \to_x C;H;\langle\langle S;L,\ell \mapsto_r \mathcal{E};e_0\rangle;\mathcal{E}[v]\rangle}\ \text{READG} \qquad \frac{L|_w(\ell) = v}{C;H;\langle\langle S;L;e_0\rangle;\mathcal{E}[!\ell]\rangle \to_x C;H;\langle\langle S;L;e_0\rangle;\mathcal{E}[v]\rangle}\ \text{READL}$$

$$\frac{}{C;H;\langle\langle S;L;e_0\rangle;\mathcal{E}[\ell := v]\rangle \to_x C;H;\langle\langle S;L,\ell \mapsto_w v;e_0\rangle;\mathcal{E}[()]\rangle}\ \text{WRITE} \qquad \frac{\ell \notin \mathrm{Dom}(H)}{C;H;\langle\cdot;\mathcal{E}[\mathbf{tref}\ v]\rangle \to_x C;H,\ell \mapsto (v,C);\langle\cdot;\mathcal{E}[\ell]\rangle}\ \text{ALLOC}$$

$$\frac{e_0 = \mathcal{E}[\mathbf{inatomic}(e)]}{C;H;\langle\cdot;\mathcal{E}[\mathbf{atomic}\ e]\rangle \to_x C+1;H;\langle\langle C;\cdot;e_0\rangle;e_0\rangle}\ \text{ATOMIC} \qquad \frac{}{C;H;\langle\langle S;L;e_0\rangle;\mathcal{E}[\mathbf{atomic}\ e]\rangle \to_x C;H;\langle\langle S;L;e_0\rangle;\mathcal{E}[e]\rangle}\ \text{NATOMIC}$$

$$\frac{\mathbf{validate}(S;L;H;C) \rightsquigarrow \mathbf{commit}(H')}{C;H;\langle\langle S;L;e_0\rangle;\mathcal{E}[\mathbf{inatomic}(v)]\rangle \to_x C+1;H';\langle\cdot;\mathcal{E}[v]\rangle}\ \text{COMMIT}$$

Figure 2. Operational Semantics ($\to_x$: common rules)

sion and then give a detailed description of the implementation in Section 5.

3.1 Syntax

Figure 1 gives the syntax of the language. Values include lambda expressions, transactional reference locations, and the unit value. Expressions include values, variables, function application, transactional dereference, update, 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 version number; we do not explicitly model a transactional reference's lock in the semantics. 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 two kinds of mappings, one which maps locations to values that were written, and one that maps locations to evaluation contexts indicating where to resume execution if the location read from is found to be invalid.

3.2 Partial Abort Operational Semantics

In order to prove the correctness of performing partial aborts, we relate our partial abort semantics 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), so we have factored out all of the common rules to a generic judgement denoted by $\rightarrow_x$, where x is then instantiated by "full", "partial", or "replay".

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. A source program e starts with the version clock set to 0, the empty heap, and a single thread $\langle \cdot; e \rangle$. A terminal program state consists of only threads that have finished evaluating their expressions to values.

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 READG rule is used for reading from a tref in the global heap that does not exist in the thread's write set, where $L|_w$ is

the log restricted to the write mappings. The location of the tref is looked up in the heap, yielding the value and version number associated with the location. This rule additionally requires that the version number associated with the location (S') is older (less than) than the thread's read version (S), which enforces part of the opacity property described in Section 2. In the conclusion of the rule, we create a read mapping in the thread's log from the location read to the current evaluation context. The READL rule simply returns the value found when looking up the location in the thread's log.

The WRITE rule records a write to a tref in the log, shadowing any previous write mappings of the location in the log. 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 [KH08], partial aborts can be used to capture many common nested transaction idioms.

The COMMIT rule is used to commit a transaction. This rule relies on the **validate** judgement given in Figure 4; for now it suffices to know that if **validate** applied to a log L yields **commit**(H'), then the log could be validated in the current program state and H' is the global heap with all locally written trefs committed. The COMMIT rule requires that **validate** yields a commit and then continues with

the current heap replaced by the one returned by validation.

The $\rightarrow_{\text{partial}}$ relation (Figure 3) describes the extension specific to performing partial aborts and simply requires the addition of two rules that also rely on the **validate** judgement. The

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

$$\frac{\textbf{validate}(S; L; H; C) \rightsquigarrow \textbf{abort}(L'; \mathcal{E}'; \ell')}{C; H; \langle\langle S; L; e_0\rangle; e\rangle \rightarrow_{\text{partial}} C+1; H; \langle\langle C; L'; e_0\rangle; \mathcal{E}'[!\ell']\rangle} \text{ ABORT\_PARTIAL}$$

$$\frac{\ell \notin \text{Dom}(L|_w) \quad H(\ell) = (v, S') \quad S' > S \quad \textbf{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \textbf{abort}(L'; \mathcal{E}'; \ell')}{C; H; \langle\langle S; L; e_0\rangle; \mathcal{E}[!\ell]\rangle \rightarrow_{\text{partial}} C+1; H; \langle\langle C; L'; e_0\rangle; \mathcal{E}'[!\ell']\rangle} \text{ READG\_PARTIAL}$$

Figure 3. Operational Semantics ($\rightarrow_{\text{partial}}$: partial abort rules)

$\boxed{\textbf{validate}(S; L; H; C) \rightsquigarrow \textbf{commit}(H') \mid \textbf{abort}(L'; \mathcal{E}'; \ell')}$

$$\frac{}{\textbf{validate}(S; \cdot; H; C) \rightsquigarrow \textbf{commit}(H)} \text{ CEMPTY}$$

$$\frac{\textbf{validate}(S; L; H; C) \rightsquigarrow \textbf{abort}(L'; \mathcal{E}'; \ell')}{\textbf{validate}(S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \textbf{abort}(L'; \mathcal{E}'; \ell')} \text{ APWRITE} \qquad \frac{\textbf{validate}(S; L; H; C) \rightsquigarrow \textbf{abort}(L'; \mathcal{E}'; \ell')}{\textbf{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \textbf{abort}(L'; \mathcal{E}'; \ell')} \text{ APREAD}$$

$$\frac{\textbf{validate}(S; L; H; C) \rightsquigarrow \textbf{commit}(H')}{\textbf{validate}(S; L, \ell \mapsto_w v; H; C) \rightsquigarrow \textbf{commit}(H', \ell \mapsto (v, C))} \text{ CPWRITE} \qquad \frac{\textbf{validate}(S; L; H; C) \rightsquigarrow \textbf{commit}(H') \quad H(\ell) = (v, S') \quad S' < S}{\textbf{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \textbf{commit}(H')} \text{ CPREAD}$$

$$\frac{\textbf{validate}(S; L; H; C) \rightsquigarrow \textbf{commit}(H') \quad H(\ell) = (v, S') \quad S' > S}{\textbf{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \textbf{abort}(L; \mathcal{E}; \ell)} \text{ AREAD}$$

Figure 4. Transactional Log Validation

ABORT_PARTIAL rule is used to partially abort a transaction. If **validate** applied to a log L yields $\textbf{abort}(L'; \mathcal{E}'; \ell')$, then the log could not be fully validated in the current program state due to a conflict, L' is the prefix of the log that could be validated, and $\mathcal{E}'$ is the continuation of the read of location ℓ' at which the conflict occurred. The ABORT_PARTIAL rule requires that **validate** yields an abort and then continues with a new read version retrieved from the global clock, the partially validated log, and the continuation ap-

plied to the read of the location. Note that the ABORT_PARTIAL rule is not syntax directed; rather, it can be applied nondeterministically at any time that the log cannot be fully validated. In practice, the ABORT_PARTIAL rule is applied when the transaction has completed and the COMMIT rule does not apply.

The READG_PARTIAL rule is used for eagerly detected conflicts. One might expect that the ABORT_PARTIAL rule suffices to capture the semantics of our full abort algorithm, where an eager abort would correspond to the inapplicability of the READG rule and, instead, applying ABORT_PARTIAL. However, it is possible for a thread to attempt to read a tref that was updated after it started its transaction while also having a fully valid log (e.g., when the log is empty and the first read is of a newly updated tref). Since the whole log is valid, the ABORT_PARTIAL rule is not applicable. In the READG_PARTIAL rule, we validate the log extended with a mapping for this attempted read of the out-of-date tref, guaranteeing that validation will discover a point of conflict and abort.

3.3 Log Validation

Figure 4 gives the rules for validating a transactional log. This judgement relates a 4-tuple containing a thread's read version, its log, the global heap, and a write version to be written into committed trefs, to a result indicating whether validation succeeded or failed. If any read tref in the log is out of date, validation fails, yielding the log prior to the invalid read and the continuation and location of the invalid read. If validation succeeds, then **validate** yields a new heap containing all of the local tref writes in the log committed to the global heap. Note that the log is validated in chronological order; this ensures that if multiple conflicts are detected, the log,

continuation, and location correspond to the earliest conflict that occurred, which is essential for the correctness of our algorithm.

The CEMPTY rule indicates that the empty log can trivially be validated. The APWRITE and APREAD rules propagate an abort through a write or read 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 mapping, by extending the committed global heap with a binding from the location to the written value and the write version; note that if a log records multiple updates to the same tref, then the latest one will shadow all earlier ones in the final committed global heap. The CPREAD rule propagates a commit through a valid read by requiring that the write version associated with the read tref in the current global heap is still older (less than) than the thread's read version. Finally, the AREAD rule initiates an abort at an invalid read when the write version associated with the read tref in the current global heap is newer (greater than) than the thread's read version; an abort result is returned with the portion of the log prior to the invalid read, the continuation of the invalid read, and the location of the invalid read.

3.4 Equivalence

The correctness of the full abort algorithm has been proven in previous work [KPH10]. 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. The semantics for the full abort algorithm consists of the common rules ($\rightarrow_x$) and two additional $\rightarrow_{\text{full}}$ rules (Fig-

ure 5. The ABORT_FULL rule is used to fully abort a transaction. Rather than making use of the log, continuation, and location returned from validation, execution proceeds with an empty log and the initial expression recorded at the beginning of the transaction.

$$\boxed{C; H; T \to_{\text{full}} C'; H'; T'}$$

$$\frac{\textbf{validate}(S; L; H; C) \rightsquigarrow \textbf{abort}(L'; \mathcal{E}'; \ell')}{C; H; \langle\langle S; L; e_0\rangle; e\rangle \to_{\text{full}} C + 1; H; \langle\langle C; \cdot; e_0\rangle; e_0\rangle} \text{ ABORT\_FULL}$$

$$\frac{\ell \notin \text{Dom}(L|_w) \quad H(\ell) = (v, S') \quad S' > S \quad \textbf{validate}(S; L, \ell \mapsto_r \mathcal{E}; H; C) \rightsquigarrow \textbf{abort}(L'; \mathcal{E}'; \ell')}{C; H; \langle\langle S; L; e_0\rangle; \mathcal{E}[!\ell]\rangle \to_{\text{full}} C + 1; H; \langle\langle C; \cdot; e_0\rangle; e_0\rangle} \text{ READG\_FULL}$$

Figure 5. Operational Semantics ($\to_{\text{full}}$: full abort rules)

$$\boxed{C; H; T \to_{\text{replay}} C'; H'; T'}$$

$$\frac{\ell \notin \text{Dom}(L|_w) \quad H(\ell) = (v, S') \quad S' > S}{C; H; \langle\langle S; L; e_0\rangle; \mathcal{E}[!\ell]\rangle \to_{\text{replay}} C; H; \langle\langle S; L, \ell \mapsto_r \mathcal{E}; e_0\rangle; \mathcal{E}[v']\rangle} \text{ READG\_REPLAY}$$

Figure 6. Operational Semantics ($\to_{\text{replay}}$: replay relation)

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

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

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

Figure 7. Thread Pool WellFormed Judgement

The READG_FULL rule is used for eagerly detecting conflicts similar to READG_PARTIAL except that a full abort takes place and there is no need for validation.

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

where $\mathrm{Done}(T)$ 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 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 ($\rightarrow_x$) and one additional $\rightarrow_{\text{replay}}$ rule (Figure 6). The READG_REPLAY rule is very similar to the READG rule except that the read is of an out-of-date tref; in this case, we allow the thread to continue with a value that has been "pulled out of thin air" (although, to be used in a derivation of the well-formedness judgement, the READG_REPLAY rule will necessarily choose the value of the read found in the log of the thread state that it is trying to recreate). This rule makes it easy to show that well-formedness is preserved by the partial abort step relation ($\rightarrow_{\text{partial}}$); 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 READG_REPLAY rule. With the WellFormed judgement, we can prove the forward direction of Theorem 1 using the following theorem:

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

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

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

Figure 8. Thread Pool AheadOf Judgement

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

Proof Sketch. By induction on the derivation of $C; H; T \rightarrow^*_{\text{partial}} C'; H'; T'$ and case analysis of the last $\rightarrow_{\text{partial}}$ step taken. The only interesting cases are the ABORT_PARTIAL and READG_PARTIAL rules. In these cases, 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 READG_REPLAY rule, since the partially-aborted thread has been restarted with a valid log. □

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 the future of" the full abort program state. To do so, we give an "ahead-of" judgement in Figure 8 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 ($\rightarrow_{\text{replay}}$) and specifies that a non-transactional thread is only related to an identical non-transactional thread. Therefore, if $\text{AheadOf}(C; H; T_{\text{f}}; T_{\text{p}})$ 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)**.** $\forall C\ C'\ H\ H'\ T_p\ T'_p\ T_f\ T'_f$, *if* $\text{AheadOf}(C; H; T_f; T_p)$ *and* $C; H; T_f \rightarrow^*_{\text{full}} C'; H'; T'_f$, *then* $C; H; T_p \rightarrow^*_{\text{partial}} C'; H'; T'_p$ *and* $\text{AheadOf}(C'; H'; T'_f; T'_p)$.

Proof Sketch. By induction on $C; H; T_f \rightarrow^*_{\text{full}} C'; H'; T'_f$ and case analysis of the last $\rightarrow_{\text{full}}$ step taken. The most interesting case is the COMMIT rule. 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 AheadOf$(C; H; \langle\langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(v)]\rangle; T_p)$, we know that T_p is of the form: $\langle\langle S; L'; e_0\rangle; e'\rangle$ and $C; H; \langle\langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(v)]\rangle \rightarrow^*_{\text{replay}} C; H; \langle\langle S; L'; e_0\rangle; e'\rangle$, 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 $T_p = \langle\langle S; L; e_0\rangle; \mathcal{E}[\textbf{inatomic}(v)]\rangle$, allowing it to also commit in the partial abort semantics. □

Note that many cases and supporting lemmas are left out for brevity and that the proof sketches provided are only meant to give the reader a high level intuition as to how the details of the proof fit together. Full details about the proof can be found in the Coq formalization at `http://www.cs.rit.edu/~ml9951/icfp15-coq-proofs.tar`.

4. Manticore

We have implemented our partial-abort extension in the context of the Manticore project [FFR+07]. 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 [MTHM97] extended with implicit fine-grain parallelism [FRRS11] and with explicit CML-style concurrency [Rep99, RRX09]; 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 ML-RISC code-generation framework [GGR94]

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 is written in C, the majority of the Manticore runtime system, including the scheduling infrastructure [FRR08] 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 [Wan80, HFW84, Rep89, Ram90, Shi97, Rep99]; they serve as the foundation for the Manticore scheduling infrastructure [FRR08] 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*.

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 [DF92] 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 con-

tinuations, 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 [BR09] and aggressive inlining [BFL$^+$14].

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 [App92]. 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 [BWD96] and escaping [RP00, FR02] 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 [DL93, DG94] and the Appel semi-generational collector [App89] and is described more fully in previous work [ABFR11]. 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

[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.

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.

5. Implementation

The STM library is implemented in the BOM IR, which as previously mentioned includes mutable references and first-class contin-

uations. This substantially simplifies the implementation, requiring less than 400 lines of code and 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 tvar that are read from and written to by the programmer. The second element is for the version number of the tref and the last element serves as a lock for the tref. A tref is locked by writing the thread's read version into the tref; a thread can use the lock value to determine if it has already acquired a given lock.

Each thread maintains three pieces of information within its thread local storage: 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.

5.1 Writes

Writing to a tref is the simplest operation. Each time a tref is written, an entry is added to the write set that records both the tref being written to and the value being written. Note that we do not perform destructive updates in the write set when the same tref is written to more than once during the transaction. This is necessary to properly restore the state of the write set when performing partial aborts (see Section 5.4).

5.2 Reads

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 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 9. 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.

5.3 Commit

When committing a transaction, a thread first acquires the lock for every tref recorded in the write set. Next, a write version is acquired

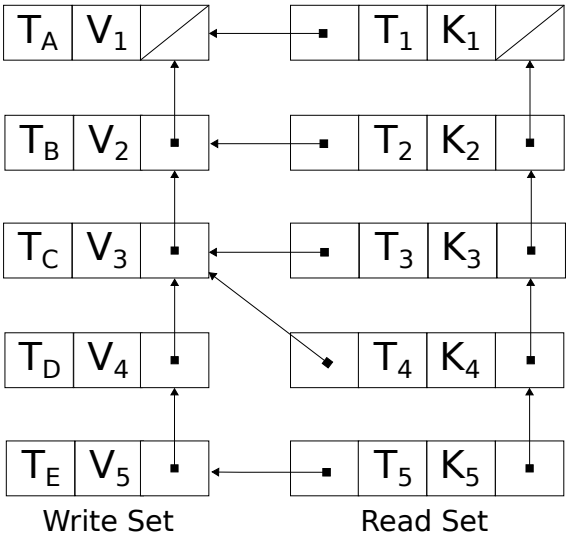

Figure 9. Layout of Read/Write sets

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, and release the tref lock.

5.4 Read Set Validation

When validating the read set, whether after detecting an eager conflict or during a commit, the validation is performed with respect to the thread's read version. 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 9)
- Update the read version to the version number received prior to validation
- Throw to `k`

If, after traversing the entire read set, the checkpoint is of the form `NONE`, then we do one of two things. If we validated the read set because a transaction is trying to commit, then we continue with the commit phase by pushing the thread's write set into the global store. If we validated because a conflict was eagerly detected in a read, then we update the thread's read version to the version number

acquired before validation and continue with the transaction; the value read from the tref during the eager conflict detection is now likely to be valid with respect to the new read version.

There is an interesting connection to be made here with a previously proposed optimization of similar full-abort STM implementations known as timebase extension [RFF07]. In that work, the authors propose to validate the read set every time a tref is found to be out of date when reading. If the entire read set is still valid, then

	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 10. Linked List Stats (Full Abort vs. Partial Abort)

the transaction can continue with a new read version that the thread acquires before performing validation. If validation fails, then the transaction aborts and restarts from the beginning. This optimization falls out naturally from performing partial aborts when a conflict is detected eagerly and generalizes the previously proposed method by being able to salvage a portion of the transaction if the entire read set cannot be validated.

5.5 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 this 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 10 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.

5.6 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 Section 6.

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 11. There still exists a long path, which passes through every node in the linked list; however, there is also now a short path which only passes through items in the linked list that contain a continuation. Lastly, each

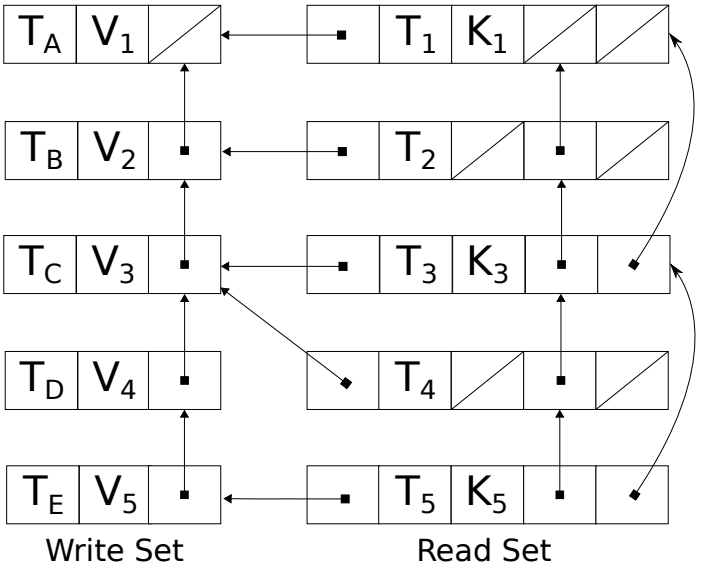

Figure 11. Skip List Representation of Read Set

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 11 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.

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 10, 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.

5.7 Chronologically Ordered Read Sets

One downside to the linked list 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

	FA Time	PA Time	Change in Time	FA Aborts	PA Aborts	PA % Partial Aborts	Change in Aborts
Delaunay Mesh	6.54	6.49	-0.73%	124,105.26	90,193.74	16.43%	-27.33%
Labyrinth	17.26	11.79	-31.67%	193.02	157.22	83.64%	-18.55%
Linked List	9.19	6.72	-26.94%	11,538.46	7,996.62	87.9%	-30.7%
Red Black Tree	8.92	10.03	+12.55%	3,684.88	4,557.72	93.73%	+23.68%
Vacation	2.99	2.45	-18.00%	12,040.56	10,970.96	88.61%	-8.89%
KMeans	3.34	3.41	+2.08%	28,799.38	10,537.1	0.00%	-63.42%
STMBench7	6.53	6.12	-6.33%	150.02	236.67	3.77%	+57.75%
Sudoku	2.9	2.63	-9.1%	18,820.24	17,946.46	86.47%	-4.65%

Figure 12. Benchmark Results (FA corresponds to Full Abort and PA Corresponds to Partial Abort with Bounded Continuations)

fine for the full abort implementation, since it is only concerned with whether a conflict exists or not; thus, if 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.

The ability to append onto the end of a linked list could potentially speed up the read set validation process substantially, by maintaining a read set that is in chronological order. Unfortunately, the Manticore heap layout precludes us from doing this efficiently.

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 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.

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 10.04.2 LTS, kernel version 2.6.32-67. 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 benefit of partial aborts, we have selected eight benchmarks typically used in evaluating STM; many come from the STAMP benchmark suite [CCKO08]. Benchmarks were chosen to provide a wide spectrum of workloads including long transactions, short transactions, and a mix of the two. In this evaluation, we only consider the partial abort implementation that includes the bounded continuation optimization described in Section 5.6, where we limit the number of continuations to 20 for each transaction.

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 4,000 elements and then each thread performs 3,000 operations, consisting of queries, insertions, and deletions with a ratio of 2:4:1 [SLM08]. This benchmark consists of very long transactions, which present excellent opportunities for partial aborts.

Delaunay Mesh (STAMP) Delaunay Mesh implements Ruppert's algorithm for Delaunay mesh refinement. The mesh is represented as a graph of triangles, where each triangle is represented as a tref. Additionally, there is a shared work queue that is protected by a tref. This benchmark consists of both very short transactions (enqueuing/dequeuing from the work queue) and medium-short transactions (refining the mesh).

Labyrinth (STAMP) Labyrinth implements Lee's parallel routing algorithm [WKL07]. 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.

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.

Sudoku Sudoku implements a concurrent sudoku puzzle solver [PSS+08] for 16 X 16 sized puzzles. Each cell in the puzzle is protected by a tref. In each iteration of the solving process the board is pruned, where one thread prunes across the rows, one across the columns, and one across the boxes. Threads can potentially prune the same element of the board, which makes use of transactions. This benchmark has medium length transactions.

STMBench7 STMBench7 [GKV07] is a benchmark specifically

designed for evaluating transactional memory systems. The benchmark simulates an in-memory object graph for a CAD/CAM system, where threads perform randomly selected operations containing different transactional workloads (long, short, large write sets, etc.).

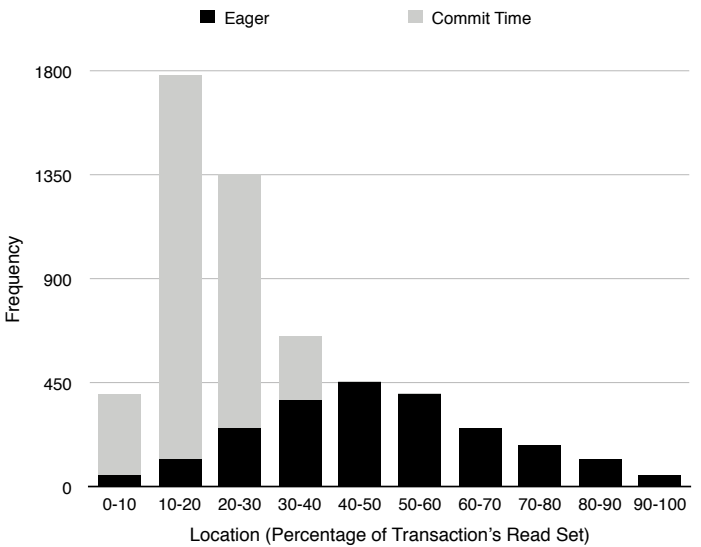

Figure 13. Red Black Tree Partial Abort Positions

6.2 Benchmark Results

Figure 12 presents results for the previously described benchmarks. Each benchmark is the average of 50 executions, each utilizing four threads. In terms of number of aborted transactions, partial abort performs better in the majority of cases, aborting more transactions only on Red Black Tree and STMBench7. Interestingly, partial abort reduces execution time by 6.33% on STMBench7 despite the fact that it aborts 57.75% more transactions and only 3.77% of the aborts were partial aborts. The reason for this is that some transactions in this benchmark are very large, so even a few partial aborts can have dramatic effects on execution time.

As is expected, the benchmarks that contain many large transactions benefit the most from performing partial aborts. Linked List and Labyrinth perform substantially better when partial aborts are performed, decreasing execution time by 26.94% and 31.67%, respectively. Most of the benchmarks that have medium length transactions also perform quite well, decreasing executing time 6%-18%, with the exception of Red Black Tree.

After looking into why the performance for Red Black Tree was so poor, we found that the position that we typically partially-abort to is very early on in the transactions. 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 full abort implementation will detect the conflict early on; however, when performing partial aborts, we must traverse the entire read set

in order to find the earliest safe checkpoint. Thus, we pay a large overhead in finding a safe place to abort to, but we get little benefit out of the partial abort since it occurs so close to the beginning of the transaction.

Figure 13 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

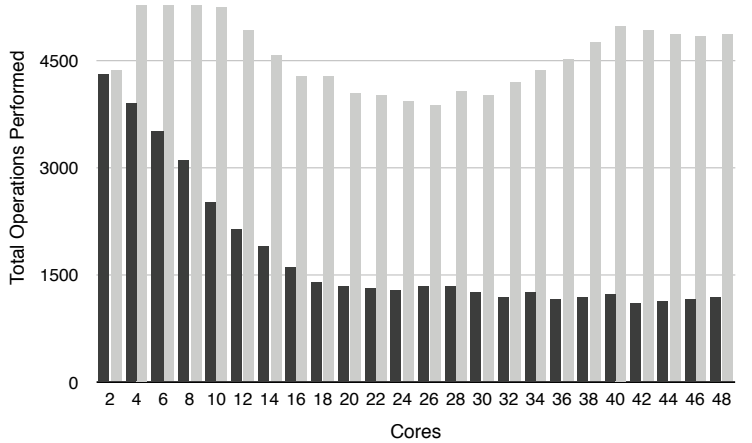

Figure 14. Total Operations

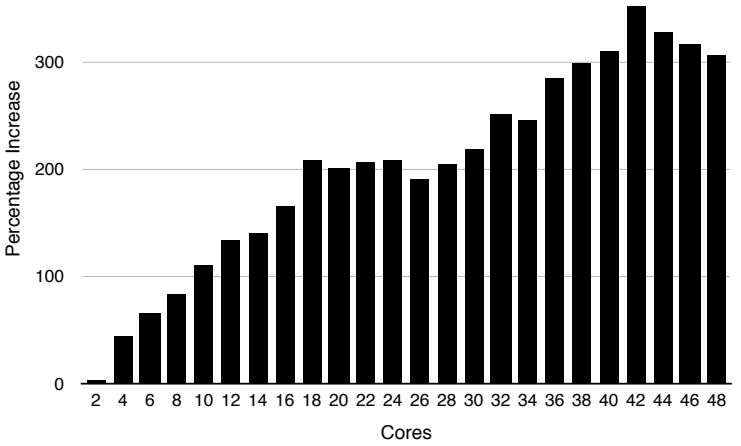

Figure 15. Percentage Throughput Increase Relative to Full Abort

benefit than a partial abort performed at commit time. We believe that performance would improve dramatically if a chronologically ordered skip list could be used for the read set, allowing validation to efficiently take place.

For Delaunay Mesh Refinement, we see a minor speedup of 0.73%. This unimpressive improvement can be attributed to the mix of very short and medium-short transactions. When enqueuing/dequeueing from the work queue, there is no chance of partially aborting, which also drives up the number of full aborts for this benchmark: of the total aborts, only 16.43% are partial aborts.

KMeans also exhibits poor performance; however, this is to be expected as there is zero opportunity for partially aborting any transactions. This benchmark was used to serve as a baseline in order to see what kind of overheads are introduced from keeping the extra information in the read set. Interestingly enough, KMeans performs fewer aborts under the partial abort implementation despite the fact that it is not partially aborting anything. This can be attributed to the fact that if a thread detects an eager conflict and is able to validate its entire read set, it can continue with the transaction without aborting. Since read sets are very small in this benchmark, this happens quite often.

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 10 seconds and the total number of operations completed by the threads working in the second half of the list is recorded.

Figures 14 and 15 contain the results of this experiment, giving the number of completed operations performed by second-half threads (Figure 14) and the percentage increase in throughput (measured by completed operations) relative to full abort (Figure 15). Again, 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.

Clearly, scalability is poor for the linked list benchmark; however, this is to be expected. This is an inherently sequential application, so, as contention gets higher, the number of aborted transactions goes up. That said, the partial abort implementation does perform much better than the full abort implementation. For full abort, the best total throughput occurs when there is only one thread working on each half of the linked list and degrades substantially as additional threads are added. The partial abort implementation performs better than full abort across the board on all configurations. Furthermore, there are five instances (cores 40-48), where the percentage increase in throughput exceeds 300%, maxing out at 351%.

7. Related Work

The most closely related work is [KH08], where the authors first proposed partially aborting transactions. The main difference is in the implementation of partially aborting transactions. Here, the authors need to perform stack copying in order to safely revert

control to a checkpoint in the event of a violation. In our 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 performs well across many of our benchmarks.

Gupta et al. also explored checkpointing transactions in [GSA10]. 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.

Nested transactions have been proposed in a number of variations [MBM$^+$06, NMAT$^+$07, HS07], 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 16 shows how 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

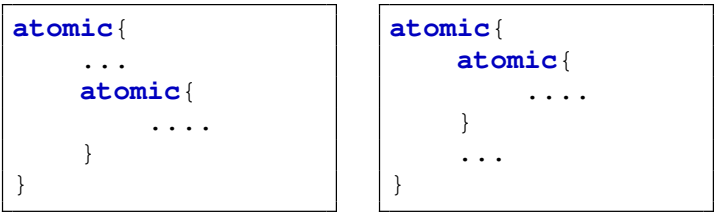

Figure 16. Common Nested Transactional Idioms

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 [RFF07] has been used in many recently proposed STM systems [FFR08, SDMS09, RFF06], 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. As was mentioned in Section 5.2, 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.

Ziarek et al. proposed a language abstraction called the *stabilizer* [ZSJ06], 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 transitive 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 [LWFA08]. In this 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.

8. Conclusion

In this paper we 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 Section 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 perfor-

mance. 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.

Although the work presented here is based on the Transactional Locking II algorithm, our partial abort extension could easily be added on top of other lazy versioning STMs. In fact, we currently have a preliminary version of NoRec [DSS10] that has been extended with partial aborts. We leave it to future work to explore adding partial aborts to STMs that use encounter-time locking as opposed to lazy versioning.

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 [Rai10], 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.

We are also interested in exploring user defined checkpointing policies in the future. Capturing continuations uniformly works well for the majority of benchmark applications that we presented in this work, however, certain applications, such as red black tree have odd conflict patterns that the programmer could characterize in a more ad hoc manner.

Acknowledgments

This research is supported by the National Science Foundation under Grants CCF-0811389 and CCF-101056

References

[ABFR11] Auhagen, S., L. Bergstrom, M. Fluet, and J. Reppy. Garbage Collection for Multicore NUMA Machines. In *MSPC 2011: Memory Systems Performance and Correctness*, San José, California, USA, June 2011. ACM.

[App89] Appel, A. W. Simple generational garbage collection and fast allocation. *Software – Practice and Experience*, **19**(2), 1989, pp. 171–183.

[App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, Cambridge, England, 1992.

[BBA15] Baldassin, A., E. Borin, and G. Araujo. Performance implications of dynamic memory allocators on transactional memory systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*

(PPoPP '15), San Francisco, CA, February 2015. ACM, pp. 87–96.

[BFL+14] Bergstrom, L., M. Fluet, M. Le, J. Reppy, and N. Sandler. Practical and effective higher-order optimizations. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functonal Programming ICFP '14*, Gothenburg, Sweden, September 2014. ACM, pp. 81–93.

[BR09] Bergstrom, L. and J. Reppy. Arity raising in manticore. In *21st International Workshop on the Implementation of Functional Languages (IFL '09)*, Lecture Notes in Computer Science. Springer-Verlag, September 2009, pp. 90–106.

[BWD96] Bruggeman, C., O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the SIGPLAN Conference on Programming Lan-*

guage Design and Implementation (PLDI '96). ACM, May 1996, pp. 99–107.

[CCKO08] Cao Minh, C., J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multiprocessing. In *IISWC '08*, September 2008.

[DF92] Danvy, O. and A. Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, **2**(4), 1992, pp. 361–391.

[DG94] Doligez, D. and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages (POPL '94)*, Portland, Oregon, United

States, January 1994. ACM, pp. 70–83.

[DL93] Doligez, D. and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages (POPL '93)*, Charleston, South Carolina, United States, January 1993. ACM, pp. 113–123.

[DR14] Diegues, N. and P. Romano. Time-warp: Lightweight abort minimization in transactional memory. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '14)*, Orlando, FL, February 2014. ACM, pp. 167–178.

[DSS06] Dice, D., O. Shalev, and N. Shavit. Transactional locking ii. In *Proceedings of the 20th International Distributed Computing Conference*, vol. 4167 of *Lecture Notes in Computer Science*, Stockholm, Sweden, 2006. Springer-Verlag, pp. 194–208.

[DSS10] Dalessandro, L., M. F. Spear, and M. L. Scott. Norec: Streamlining stm by abolishing ownership records. In *PPoPP '10*, Bangalore, India, 2010. ACM, pp. 67–78.

[FFR+07] Fluet, M., N. Ford, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Status Report: The Manticore Project. In *Proceedings of the 2007 ACM SIGPLAN Workshop on ML*. ACM, October 2007, pp. 15–24.

[FFR08] Felber, P., C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '08)*, Salt Lake City, UT, February 2008. ACM, pp. 237–246.

[FR02] Fisher, K. and J. Reppy. Compiler support for lightweight

concurrency. *Technical memorandum*, Bell Labs, March 2002. Available from `http://moby.cs.uchicago.edu/`.

[FRR08] Fluet, M., M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming ICFP '08*, Victoria, BC, Canada, September 2008. ACM, pp. 241–252.

[FRRS11] Fluet, M., M. Rainey, J. Reppy, and A. Shaw. Implicitly-threaded parallelism in Manticore. *Journal of Functional Programming*, **20**(5–6), 2011, pp. 537–576.

[GGR94] George, L., F. Guillame, and J. Reppy. A portable and optimizing back end for the SML/NJ compiler. In *Fifth International Conference on Compiler Construction*, April 1994, pp. 83–97.

[GK08] Guerraoui, R. and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08*, Salt Lake City, UT, USA, 2008. ACM, pp. 175–184.

[GKV07] Guerraoui, R., M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, Lisbon, Portugal, 2007. ACM, pp. 315–324.

[GSA10] Gupta, M., R. K. Shyamasundar, and S. Agarwal. Clustered checkpointing and partial rollbacks for reducing conflict costs in stms. *International Journal of Computer Applications*, **1**(22), 2010, pp. 82–87.

[HFW84] Haynes, C. T., D. P. Friedman, and M. Wand. Continuations and coroutines. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM,

August 1984, pp. 293–298.

[HM93] Herlihy, M. and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*, San Diego, California, USA, 1993. ACM, pp. 289–300.

[HS07] Harris, T. and S. Stipic. Abstract nested transactions. In *TRANSACT 2007*, January 2007.

[KH08] Koskinen, E. and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA '08*, Munich, Germany, 2008. ACM, pp. 160–168.

[KPH10] Koskinen, E., M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *Conference Record of the 37th Annual ACM Symposium on Principles of Programming Languages (POPL '10)*, Madrid, Spain, 2010. ACM, pp. 19–30.

[LWFA08] Ley-Wild, R., M. Fluet, and U. A. Acar. Compiling self-adjusting programs with continuations. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming ICFP '08*, Victoria, BC, Canada, September 2008. ACM, pp. 321–334.

[MBM+06] Moravan, M. J., 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*, San Jose, California, USA, 2006. ACM, pp. 359–370.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[NMAT+07] Ni, Y., 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*, San Jose, California, USA, 2007. ACM, pp. 68–78.

[PSS+08] Perfumo, C., N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): Dissecting Haskell STM applications on a many-core environment. In *Proceedings of the 5th Con-*

ference on Computing Frontiers (CF '08), Ischia, Italy, May 2008. ACM, pp. 67–78.

[Rai10] Rainey, M. *Effective Scheduling Techniques for High-Level Parallel Programming Languages*. Ph.D. dissertation, University of Chicago, August 2010. Available from `http://manticore.cs.uchicago.edu`.

[Ram90] Ramsey, N. Concurrent programming in ML. *Technical Report CS-TR-262-90*, Department of Computer Science, Princeton University, April 1990.

[Rep89] Reppy, J. H. First-class synchronous operations in Standard ML. *Technical Report TR 89-1068*, Department of Computer Science, Cornell University, December 1989.

[Rep99] Reppy, J. H. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

[RFF06] Riegel, T., P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Distributed Computing Conference*, vol. 4167 of *Lecture Notes in Computer Science*, Stockholm, Sweden, 2006. Springer-Verlag, pp. 284–298.

[RFF07] Riegel, T., C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *SPAA '07*, San Diego, California, USA, 2007. ACM, pp. 221–228.

[RP00] Ramsey, N. 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`, November 2000.

[RRX09] Reppy, J., C. Russo, and Y. Xiao. Parallel Concurrent ML. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming ICFP '09*, Edinburgh, Scotland, UK, August–September 2009. ACM, pp. 257–268.

[SDMS09] Spear, M. F., L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09*, Raleigh, NC, USA, 2009. ACM, pp. 141–150.

[Shi97] Shivers, O. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations (CW '97)*. ACM, January 1997.

[SLM08] Sulzmann, M., E. S. Lam, and S. Marlow. Comparing the performance of concurrent linked-list implementations in

haskell. In *DAMP '09*, Savannah, GA, USA, 2008. ACM, pp. 37–46.

[ST95] Shavit, N. and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, Ottowa, Ontario, Canada, 1995. ACM, pp. 204–213.

[Wan80] Wand, M. Continuation-based multiprocessing. In *Conference Record of the 1980 ACM Conference on Lisp and Functional Programming*. ACM, August 1980, pp. 19–28.

[WKL07] Watson, I., C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *PACT '07*. IEEE Computer Society, 2007, pp. 388–398.

[ZHCB15] Zhang, M., J. Huang, M. Cao, and M. D. Bond. Low-overhead software transactional memory with progress guarantees and strong semantics. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP '15)*, San Francisco, CA, February 2015. ACM, pp. 97–108.

[ZSJ06] Ziarek, L., P. Schatz, and S. Jagannathan. Stabilizers: a modular checkpointing abstraction for concurrent functional programs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming ICFP '06*, Portland, Oregon, USA, September 2006. ACM, pp. 136–147.