TardisTM: Incremental Repair for Transactional Memory

Daming D. Chen
Carnegie Mellon University
ddchen@cs.cmu.edu

Phillip B. Gibbons
Carnegie Mellon University
gibbons@cs.cmu.edu

Todd C. Mowry
Carnegie Mellon University
tcm@cs.cmu.edu

Abstract

Transactional memory (TM) provides developers with a transaction primitive for concurrent code execution that transparently checks for concurrency conflicts. When such a conflict is detected, the system recovers by aborting and restarting the transaction. Although correct, this behavior wastes work and inhibits forward progress.

In this paper, we present TardisTM, a software TM system that supports repairing concurrency conflicts while preserving unaffected computation. Our key insight is that existing conflict detection mechanisms can be extended to perform incremental transaction repair, when augmented with additional runtime information. To do so, we design a mechanism for localizing conflicts back to transactional program points, define the semantics for optional repair handler annotations, and extend the conflict detection algorithm to ensure all repairs are completed. To evaluate our system, we characterize the benefit of repair on a set of benchmark programs; we measure up to 2.95x speedup over mutual exclusion, and 93% abort reduction over a baseline software TM system that does not support repair.

CCS Concepts: • Computing methodologies → Concurrent computing methodologies; • Computer systems organization → Multicore architectures.

Keywords. software transactional memory, program repair, incremental computation

1 Introduction

Transactional memory (TM) provides developers with first-class transactional regions that guarantee atomicity, consistency, and isolation, avoiding the drawbacks of low-level concurrency control primitives [21]. These guarantees are ensured by the transactional runtime, which upon detecting a concurrency conflict, will abort, roll back, and restart a transaction. However, this behavior discards all work already performed within a transaction, which impedes forward progress and decreases performance. Transactions that are more likely to encounter conflicts, such as those that are high-contention or long-running, are particularly affected.

Prior Work Helps But Still Aborts/Restarts: Prior work has developed various orthogonal strategies to mitigate this problem; namely, contention management, transaction scheduling, abort impact reduction, and conflict reduction. Nested transactions [29] reduce the scope of aborts to that of the innermost encapsulating transaction, but require strict nesting and are frequently flattened when unsupported. Early release [20], abstract locking [31], and transactional boosting [19] reduce conflicts by bridging the semantic gap between abstract datatypes and concrete implementations, where multiple implementation-defined memory states can correspond to the same abstract state (e.g., multiple lists of elements with different ordering can represent the same unordered set). Taken further, semantic commutativity [48] observes that certain abstract operations can be reordered; e.g., insertion into a linked list, increment of an integer counter, etc. However, these strategies only mitigate certain transaction conflicts, at the cost of a completely different TM runtime. Both transactional boosting and abstract locking require abstract datatypes to inform the runtime of inverse abstract operations, or to implement custom locking, which obviates much of the benefits of TM. All these prior approaches still rely on aborting and restarting.

Our Approach: Incremental Repair: In contrast, we propose TardisTM, a software TM system that supports incremental repair of conflicting transactions. Using repair annotations, TardisTM can safely resume transaction execution, ensuring forward progress and reducing wasted work. As an example, consider the simplified array-based microbenchmark shown in Listing 1, in which two concurrent transactions can conflict when one commits after incrementing the stored array value, and the other has read the now-stale previous value.
We design our repair mechanism as an extension of existing software TM [42]. In this section, we provide a brief overview, and show a commit-time write-back design in Figure 1.

**Conflict Detection:** Time-based designs (e.g., TL2 [9]) increment a global timestamp counter when a transaction successfully completes (commits). A timestamp is associated with every global memory address, which records when its value was last updated. Each transaction stores the observed timestamp for each global memory read in a local *read set*. During *read validation*, the read set is checked for stale timestamps, and if one is detected, then a *data conflict* occurs.

**Timestamp Extension** [14, 36]: To minimize validation frequency, each transaction tracks a local *validity threshold* (Figure 1), which is initialized to the global timestamp counter when the transaction starts. Read validation is performed only if a transactional read observes a newer timestamp, or attempts to commit with a stale validity threshold. If successful, the transaction can update its validity threshold to the validation-time global timestamp counter.

**Memory Locking:** Transaction atomicity is ensured by two-phase locking [13], which uses an additional global lock array. A predefined surjective function maps memory addresses to locks, which can also store the corresponding timestamps. In a word-based design, multiple memory words may map to the same lock, resulting in *lock aliasing* and allowing *spurious conflicts* (e.g., Tnystm [14], SwissTM [12], McRt-STM [40]). These locks can be acquired eagerly as memory is accessed (*encounter-time*), or lazily when a transaction attempts to commit (*commit-time*). If a lock is already held by another transaction, then a *lock conflict* occurs.

**Transactional Writes:** A local *write set* is used to track transactional writes. Under eager *write-through*, writes are performed immediately, and the write set is used as an *undo log* in case the transaction aborts. Conversely, under lazy *write-back*, writes are delayed until successful commit, and the write set is used as a *redo log*. This also requires reads to check for pending writes in the write set.

**Dynamically-Allocated Memory:** Transactional semantics can be implemented using an epoch-based *garbage collector*. Memory is queued for deallocation if it is allocated but the transaction aborts, or if it is deallocated and the transaction commits. To avoid a *use-after-free* violation by other ongoing transactions, conflicts are forced by incrementing all corresponding lock timestamps, and deallocation is delayed until all concurrent transactions have finished.

### 3 Design: TardisTM

We first discuss the design principles of TardisTM in §3.1, before summarizing our design in §3.2, and detailing our contributions in §3.3 - §3.7.

#### 3.1 Principles

**Precision:** We want to preserve transaction progress without discarding non-conflicting computation, which requires finer-grained tracking to identify data and control-flow dependencies involving stale reads. We achieve this with first-class *abstract operations* and read set *origin tracking*, which resemble lightweight dynamic *program slicing* [49].

---

**Listing 1.** A simplified array-based microbenchmark. A random shared array value is read (lines 5–6), and randomly incremented if the operation is read-write (lines 7–8). Some emulated local work is performed (line 9). This is repeated for some number of operations per transaction (line 4) and transactions per thread (line 2).

```c
void work(data_t d) {
    for (int i = 0; i < d.tx_per_thread; ++i) {
        for (int j = 0; j < d.ops_per_thread; ++j) {
            unsigned idx = RAND_UNIFORM(0, d.array_size);
            int val = d.array[idx];
            if (!d.ops[j].read_only) {
                d.array[idx] = val + RAND_UNIFORM(0, 100);
            }
        }
    }
}
```

1. https://github.com/ddcc/tardinSTM
2. https://github.com/ddcc/Stamp
Isolation: We want to avoid interacting with other concurrent transactions, in order to ensure consistency and avoid additional synchronization. Each transaction repairs its local conflicts with respect to global memory, much like the rebase operation in distributed version control systems such as git [46], which requires a write-back design (§2). This also prevents cyclic dependencies from occurring. A fully-repaired transaction thus appears as if it had instantaneously executed to the same program point.

Efficiency: To minimize overhead, we want to perform repair as infrequently as possible. This favors a commit-time locking design (§2), which we found to be much simpler than an early prototype that used encounter-time locking.

Compatibility: Since repair annotations are optional, we want to compose with existing transactions that abort.

3.2 Overview

We build TardisTM as an extension of a word-based software TM with time-based conflict detection and commit-time locking, which we denote the baseline system. At a high-level, our approach is as follows:

1. Annotations for first-class abstract operations and repair handlers are added to existing transactional code.

2. Inside a repairable transaction (§4.3), whenever an abstract operation is executed (invoked), it is tracked in a local operation log, and its corresponding read/write set entries are tagged (§3.3). Also, the source of each read set entry (origin tracking), and the chronology of each write set entry (write history) are recorded (§3.4).

3. When a conflict is detected, incremental repair (§3.5) is performed by executing any repair handlers attached to the corresponding abstract operation. Various repair policies are available, including a replay repair that automatically rolls back and re-executes the operation invocation.

4. After a successful repair, conflict detection resumes. To ensure correctness, any additional temporal conflicts (§3.6) caused by repair must also be identified and repaired.

3.3 Conflict Localization: Abstract Operations

Recall the simplified array-based microbenchmark shown in Listing 1. Observe that under the baseline system (shown in Figure 1), when a data conflict occurs on lock #0, multiple memory addresses may correspond to the same lock (e.g. 0x1000 and 0x1004), preventing localization to a specific program point. We make a number of changes in TardisTM to resolve this, discussed below and shown in Figure 2.

First, we store observed values and memory addresses in the read set, which generally correspond to language-level variables while in scope. Next, we define first-class typed abstract operations, which annotate arbitrary code sequences. Each execution (invocation) is recorded in a local operation log, including its input arguments, return value, and parent invocation (every transaction is initialized with a dummy root). Furthermore, every read/write set entry is tagged with its corresponding invocation.

Taken together, these changes record the execution history of each transaction, ensuring that when a conflict occurs, the repair handler (§3.5) can lift memory addresses back to language-level variables at a specific point in time.

Footnote: Because variables are identified by memory address, any that are modified during repair must be address-taken and thus memory-backed. We also define an optional user-defined tag field (omitted for clarity), which can be used to disambiguate between sum types (e.g., unions in C), record the base address of a product type (e.g., structs in C), or store other implementation-defined repair information.
3.4 Dependency Resolution: Origin Tracking

Likewise, under the baseline system, reads of local writes are not tracked in the read set, and existing entries in the write set are overwritten when subsequent writes occur at the same address. This inhibits repair, because intermediate write-read dependence edges are lost, preventing detection of these stale computations. To address this problem, we perform origin tracking, which includes a mechanism for storing write history, shown in Figures 2 and 3.

First, we create an entry in the read set for each read of a local write. To disambiguate between reads of global memory and the write set, we introduce an "origin" field in the read set that specifies the source (GM or WS). As an optimization, we enable deduplication of read set entries to avoid storing multiple copies of the same read (e.g., if executed in a loop).

Next, to determine if a write set entry has been changed since it was last read (e.g., by repair), we introduce a per-entry last update timestamp for the write set that is analogous to the last update timestamp for global memory. Thus, the "timestamp" field of each read set entry is interpreted based on its origin, either with respect to a local write set entry or to a shared lock array entry. These changes affect conflict detection, which we discuss subsequently in §3.6.

Finally, instead of immediately overwriting existing write set entries, we implement a lightweight history mechanism that tracks the chronology of past writes at the same address. Since each write set entry is already associated with an abstract operation invocation (§3.3), we identify past entries by their address and abstract operation invocation. This means that an existing entry is overwritten only if both fields match, and otherwise a new entry is created and the old entry is returned.

3.5 Incremental Repair

3.5.1 Overview. When attempting incremental repair, we look up the appropriate repair handler for the conflicting abstract operation invocation. We distinguish between just-in-time and delayed repair handlers, which have different capabilities depending on whether the conflicting invocation is still executing (and thus on the stack). The former are lexical closures, which can directly modify the execution context, and complete using a continuation that allows execution to resume at any program point within the same abstract operation. In contrast, the latter must utilize a different interface, described below in §3.5.2.

Next, each repair handler must select from one of two different repair policies: replay or manual. Under a replay repair, the repair handler delegates the repair to TARDISTM, which will automatically revert all effects of the conflicting invocation on the local transaction, and re-executes it at that point in time with the same arguments. This requires that an abstract operation corresponds to an implementation function of the same type.

Subsequently, our conflict detection algorithm (§3.6) will identify any orphan or stale dependent computations as temporal conflicts that must also be repaired to avoid transaction abort. Changes to non-memory-backed dependencies, such as return value, constitute a recursive conflict that precipitates repair at the parent invocation, effectively recursing up the execution stack recorded in the operation log.

3.5.2 Manual Repair. During a manual repair, the repair handler is provided with the conflict context, and given an opportunity to perform repair. We show examples of a delayed repair in Listing 1, involving a stale read at some array index. As outlined in §1, the value of the stale read is fetched (line 4–5) and updated (line 6). If a write occurred (line 8), the stale write is updated (lines 10–11).

Listing 2. A simplified manual delayed repair for Listing 1, involving a stale read at some array index. As outlined in §1, the value of the stale read is fetched (line 4–5) and updated (line 6). If a write occurred (line 8), the stale write is updated (lines 10–11).
The conflict context always includes the memory address and corresponding abstract operation invocation of the conflict. Depending on the conflict type, it may also include a reference to the stale read set entry (data conflict), or the values of the previous and current return values (recursive conflict). Each repair handler must indicate if it succeeded, and may provide a new return value. Should it fail, the transaction may have become inconsistent and must abort.

We provide repair handlers with a special interface for querying and modifying the internal state of TardisTM, including the read set, write set, and operation log. These operations, summarized in Table 1, are temporally scoped such that the affected transaction observes the current global memory without the effects of its subsequent transactions. This ensures that queries cannot retrieve its future state, and modifications cannot affect its past state—paradoxes that would violate temporal consistency.

New writes that are created by the repair handler must be carefully handled, because the origin of subsequent reads at the same address should now point to this write. In the event that the conflicting invocation has already performed a write at this address, we simply abort the repair because their chronology cannot be automatically resolved. After eagerly correcting the “origin” field of subsequent reads, we deliberately invalidate their observed timestamp to ensure subsequent identification by conflict detection (§3.6).

Note that bugs or unsafe behavior within repair handlers can violate transaction consistency. For example, non-transactional reads and writes will evade the temporal scoping provided by our design. Likewise, failure to update a stale write will prevent detection of its subsequent stale reads.

### 3.6 Temporal Conflict Detection

To ensure the correctness and convergence of incremental repair, we need to perform each individual repair in their original execution order. Because repair occurs whenever a conflict is detected, this means that a particular ordering of conflict detection is needed. In contrast, the baseline system aborts and restarts when any conflict is detected.

We resolve this by imposing chronological ordering on the read set, and limiting conflict detection to read validation, which iterates sequentially through the read set. During

### 3.7 Dynamically-Allocated Memory

Transactional operations on dynamically-allocated memory may also need to be repaired. To support this, we tag queued memory operations (§2) with their corresponding abstract operation invocation, and extend the repair handler interface to include these operations (Table 1). Thus, deallocations can be reverted by simply removing the queued request. But, allocations cannot be reverted by deallocating immediately, because they could be reused by the memory allocator and alias with an existing unrepaired read/write set entry. Instead, reverted allocations must be queued for the garbage collector.

### 4 Implementation

We implement TardisTM on top of TinySTM [14], an existing software TM written in the C programming language. Our changes amount to approximately an additional 5.7 kloc, computed using cloc [8].

### 4.1 Abstract Operations

Abstract operations are dynamically registered with TardisTM at startup using a function with the constructor attribute, a compiler extension supported by both the GNU C Compiler (GCC) and Clang. A macro is used to generate the body of this function for all abstract operations defined in the current compilation unit. Manual calls are inserted to record input arguments and return value for each abstract operation.
4.2 Just-in-Time Repair

Just-in-time repair handlers (§3.5.1) are implemented using nested functions, a GCC-specific feature for lexical closures [3]. Because these are generated as executable stack trampolines, we ensure that the corresponding invocation is still executing by recording and subsequently invalidating the address of the nested function in the operation log. To perform automatic replay, we record the implementation function for each abstract operation and invoke it using the 1libffi [15] library, which can dynamically generate typed function calls for the platform-specific calling convention.

Continuations are implemented using another GCC-specific extension, local labels, which allow nested functions to immediately return from any intermediate stack frames and goto the code at the label. Just-in-time repair handlers must first call a special cleanup function to fix-up the internal state of TardisTM before doing so; e.g. locks may need to be released if the transaction was committing.

4.3 Adaptive Execution

To reduce overhead, we proactively check whether the current abstract operation invocation is repairable. If not, we disable operation logging (§3.3), as well as origin tracking and write history (§3.4).

5 Evaluation

To demonstrate the effectiveness of transaction repair, we evaluate TardisTM on a set of benchmark programs: array (Listing 1), and the Stanford Transactional Applications for Multi-Processing [28] (STAMP).\(^8\) We manually developed repair annotations for various operations on linked lists, hashtables, queues, and red-black trees, as well as program-specific logic and data structures. Most of these were for STAMP’s internal data structure library, which are used by multiple individual benchmarks.

We compare against TinySTM, baseline TardisTM without all repair-related functionality\(^8\) (TardisTM-None), lock-based mutual exclusion (Mutex), as well as GCC’s compiler-based software system (GCC) and compiler-based hybrid [7] (GCC-RTM), which utilizes hardware Intel Restricted Transactional Memory [17] (RTM).

Our benchmarks were performed on a system with a Samsung 850 EVO 500 GB SSD, 64 GB DDR4 ECC RAM per socket, and 2x Intel Xeon E5-2683v4 CPUs at 2.6 GHz, running Debian 10 with Linux kernel 4.12.6-1 and GCC 8.3.0-6.\(^9\) All experiments were averaged over 10 runs, with speedup error bars set to one standard deviation.

5.1 Benchmark Characterization

We characterize the repair potential for these benchmarks by measuring transaction cycles (using rdtscp) under TardisTM-None, and computing the following metrics:

\[
\text{Average Wasted Work} = \log_2 \left( \frac{\sum \text{Aborted TX Cycles}}{\text{# Threads}} \right)
\]

\[
\text{Abort Rate} = \frac{\# \text{Aborts}}{\text{# Aborts} + \# \text{Commits}}
\]

We show the results in Figures 4a and 4b, respectively, which we use to roughly categorize our benchmarks as follows: High/High, where we expect significant speedups (§5.2), Mixed, where we expect moderate speedups (§5.3), and Low/Low (omitted), which are summarized in Table 2. Observe that for each benchmark, higher thread count increases both metrics, due to an increased probability of conflict.

5.2 Case #1: High Wasted Work, High Abort Rate

Our array benchmark (Listing 1) falls into this category. Conflicts in array occur when concurrent transactions access and modify values at the same array index. Instead of aborting, TardisTM uses a manual delayed repair handler to perform incremental repair, shown earlier in Listing 2.

As shown in Figure 5a, TardisTM achieves a speedup of up to 1.62x compared to Mutex, and is consistently the fastest among the compared systems on array. Surprisingly, TardisTM-None, GCC, and TinySTM almost always performed worse than Mutex, likely due to frequent aborts.

\(^8\)We made some changes to the STAMP benchmarks; namely, modernizing the code using C11 standard library atomics, and padding 32-bit floating-point variables to avoid word aliasing on 64-bit systems. We also fixed numerous resource leakage and correctness bugs (§5.5), some of which were identified by previous work [23, 37]. Where necessary, we moved local variable declarations to be within repair handlers, and adjusted control-flow into single-entry single-exit regions for ease of annotation.

\(^9\)This includes the operation log, origin tracking, write history, etc.

---

Table 2. Benchmarks, characteristics, and TardisTM improvements by category.
Compared with TardisTM-None, TardisTM achieves an abort reduction of between 61% and 78% (Figure 6a) and a speedup of up to 2.78x (Figure 6b).

5.3 Case #2: Mixed Wasted Work and Abort Rate
Two benchmarks, genome and bayes, fall into this category.

**genome**: This program performs batched reassembly of textual DNA segments for genome sequencing. Fragments are inserted into a fixed-size hashtable, which is equivalent to a membership check on the bucket linked-list, followed by insertion if not present. Because the return value of the hashtable insertion is never checked, repair can be fully delegated to the underlying annotation, shown in Listing 3. Conflicts during linked-list insertion can occur at three different positions: (1) within the operation to find the predecessor, (2) when reading the predecessor, or (3) when incrementing the list size. All can be repaired by updating the predecessor if changed (line 6), reverting all reads/writes from the current invocation (line 7), and re-inserting (line 9). As shown in Figure 5b, TardisTM achieves a speedup, starting at 8 threads, of up to 1.92x compared to Mutex. In contrast, GCC and GCC-RTM perform worse than Mutex, whereas TinySTM and TardisTM-None perform better, but still worse than TardisTM. Compared with TardisTM-None, TardisTM achieves an abort reduction of between 34% and 93% (Figure 6a) and a speedup of up to 1.22x (Figure 6b).
Listing 3. Implementation and just-in-time repair for a simplified linked-list insertion (LIST_INSERT) abstract operation.

```c
void list_insert(list_t *listPtr, void *dataPtr) {
    node_t *prevPtr = NULL, *nodePtr = NULL;
    tardistmrepair_t repair(tardistmconflict_t c) {
        if (c.recursive && TARDISTM_OP_ID(c.prev_op) == LIST_PREVIOUS)
            prevPtr = c.conflict.rv.ptr;
        TARDISTM_REVERT_RW(c.current_op);
        TARDISTM_FINISH_REPAIR();
        goto insert;
    }
    prevPtr = LIST_PREV(listPtr, dataPtr); // (1)
    nodePtr = LIST_ALLOC_NODE(dataPtr);
    insert:
    nodePtr->nextPtr = prevPtr->nextPtr; // (2)
    prevPtr->nextPtr = nodePtr; // (2)
    listPtr->size += 1; // (3)
}
```

**Figure 7.** Cumulative overhead at 4 threads of TardisTM over TardisTM-None as runtime tracking features (and finally actual repair) are enabled, for each of the benchmarks.

bayes: This program performs bayesian inference by searching for the edge between variables that maximizes prediction log-likelihood [5]. The search is transactional, and may conflict when log-likelihood estimates are concurrently updated. However, because these values always increase monotonically, conflicts do not affect the best edge unless the baseline increases to make it invalid. Thus, repair typically only needs to increment the final score by the improvement.

As shown in Figure 5c, TardisTM achieves a speedup of up to 2.95x compared to Mutex. These speedups are occasionally matched by TinySTM and TardisTM-None, whereas both GCC and GCC-RTM barely perform better than Mutex.

Compared with TardisTM-None, TardisTM achieves an abort reduction of between 41% and 76% (Figure 6a) and a speedup of up to 1.26x (Figure 6b). However, these results should be taken cautiously, given the high variance and non-deterministic behavior of bayes observed by past work [37].

### 5.4 Overhead

Runtime tracking is required for repair, which imposes overhead on all transactions regardless of whether they abort or commit. To measure this, we show performance while cumulatively enabling tracking in Figure 7, using 4 threads. The results show that incremental repair improves performance when enabled (YesRepair) compared to when not enabled (NoRepair), across all but kmeans-low, labyrinth, and ssca2. However, this improvement does not necessarily exceed the overhead of origin tracking (Origin), operation logging (OpLog), and other repair-related code (NoRepair) at 4 threads. Although we do expect absolute speedups to increase with greater thread count, based on our previous figures, we observe that overall repair effectiveness is affected by a variety of factors, including repair cost, number of repairs, and the repair success rate, as all repairs must complete successfully for a transaction to avoid aborting.

### 5.5 Discussion

As shown in Figure 6a, TardisTM achieves significant reductions in abort rates across benchmarks and thread counts, compared to TardisTM-None (Figure 4b). However, this large reduction often does not result in any speedups (Figure 6b), due to repair overhead (§5.4).\(^{11}\) We do not expect speedups on the remaining benchmarks (kmeans, intruder, ssca2, vacation, and yada), due to low abort rate or low wasted work. Some are dominated by small transactions (e.g. kmeans, ssca2), where repair overhead exceeds abort cost, whereas others (e.g. intruder, yada) contain data structures that are impractical to repair incrementally, despite our automated replay repair. For example, insertion/deletion on a red-black tree can involve recursive rebalancing, which may affect all subsequent operations.

Another contributing factor is that the STAMP benchmarks have been restructured for TM. For example, intruder has been split into two transactions that pass data non-transactionally: one solely dequeues an element, whereas the other performs packet reassembly on it. Others perform non-transactional operations within transactions, which is unsupported by transactional compilers like GCC\(^{12}\), and resulted in many correctness bugs that we fixed. These include memory leaks and use-after-free bugs involving non-transactional dynamically-allocated memory operations, as well as non-transactional writes that are unobservable within transactions, or irreversible when the transaction aborts. Nevertheless, some of the programming models and data structures used by the STAMP benchmarks are suboptimal, which adversely affects scaling with increased thread count [30].

\(^{11}\)Labyrinth relies entirely on privatization and manual aborts [47], which bypasses both transactional conflict detection and our repair mechanism. Under GCC and GCC-RTM, all operations in transactions were executed transactionally.
5.6 Repair Annotations
We implemented 76 repair annotations total, of which 69 were manual repairs. Estimated sizes are shown in Table 3, with STAMP’s internal data structure library listed as \texttt{lib}. This includes debug code and disabled annotations that were difficult to automatically exclude; for example, the core repair for \texttt{array} from Listing 2 amounts to 13 loc, instead of 47 loc.

<table>
<thead>
<tr>
<th>Program</th>
<th>array</th>
<th>bayes</th>
<th>genome</th>
<th>intruder</th>
<th>kmeans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repair loc</td>
<td>47</td>
<td>189</td>
<td>61</td>
<td>232</td>
<td>130</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>labyrinth</th>
<th>lib</th>
<th>scca2</th>
<th>vacation</th>
<th>yada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Repair loc</td>
<td>N/A</td>
<td>1780</td>
<td>56</td>
<td>438</td>
<td>368</td>
</tr>
</tbody>
</table>

Table 3. Repair size upper bounds, in lines of code (loc).

6 Related Work
We provide an overview of related work on transactional conflicts, which we categorize as follows. As mentioned previously in §1, these are orthogonal to TardisTM, because none support general transaction repair.

Contention Management: A contention manager dynamically handles conflicts. Policies include waiting with exponential backoff [20], based on work performed [41], aborting based on work performed [41], and prioritizing based on earliest original start time [16].

Transaction Scheduling: A scheduling mechanism determines when transactions start or commit. Decisions can be made based on contention [50] or commit ordering [1, 35, 38].

Abort Impact Reduction: Transactions are structured to minimize abort effects. Transactional checkpoints [24] partially abort to user-defined rollback points, whereas hardware pre-abort handlers [33] partially commit and fall back to software. Nested transactions [29] scope aborts to that of the innermost affected transaction. Open nesting and closed nesting differ in the visibility of committed child transactions, whereas abstract nesting [18] may delay restart of aborted child transactions.

Conflict Reduction: The conflict detection algorithm can avoid certain conflict types. Value-based conflict detection [6, 11, 32] is impervious to timestamp conflicts from lock aliasing. Subsequent work [39] adds first-class comparison and increment primitives that support semantic-aware recomputation. RetCon [2] performs hardware symbolic recomputation using program slicing, but is limited by complex expressions or control-flow changes. Multi-version memory [10, 27, 34] allows concurrent transactions with different memory versions, but may ultimately need reconciliation.

Semantic commutativity [48] can avoid conflicts that semantically commute. These include an approach using open nesting [31], a model [25] for coarse-grained transactions, a commutativity lattice [26] for reasoning about commutativity conditions, a method [4] for retrofitting abstract locks, hardware support [51] for commutative operations, a compositional transactional data structure library [44], and datatype-level transactional semantics [22].

Alternatively, parameterized atomic blocks [45] allow hardware to ignore certain conflicts on a per-transaction basis, and early release [20] allows manual removal of read set entries, which can be useful for linear data structures [43].

7 Conclusion
In this paper, we introduced TardisTM, a software TM that supports incremental repair. We find that repair is especially useful for workloads with high abort rate and wasted work, achieving up to 1.62x speedup over mutual exclusion and 78% abort reduction over baseline, on \texttt{array}. Repair is also useful for workloads with mixed abort rates and wasted work, reaching up to 2.95x speedup over mutual exclusion and 93% abort reduction over baseline on the STAMP benchmarks.

Given the trade-offs between runtime overhead and repair capability, we are interested in exploring improvements as future work. For example, a hardware-based repair mechanism would significantly reduce the overhead of our current approach. In addition, statically pre-computed repairs using program slicing could eliminate user-defined repair handlers and runtime dependency tracking.

Acknowledgments
This work was supported in part by the National Science Foundation, and by the Department of Defense through the National Defense Science & Engineering Graduate Fellowship Program. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

We would like to thank Chris Fallin and Pratik Fegade for their suggestions, as well as the Parallel Data Lab, Guy Blelloch, Sol Boucher, Thomas Kim, and Dave Andersen, for access to additional computational resources.

References


[37] Wenjia Ruan, Yujie Liu, and Michael Spear. 2014. STAMP Need Not Be Considered Harmful. In TRANSACTIONS ON ARCHITECTURAL CODE OPTIMIZATION 14. ACM.


[43] Travis Skare and Christos Kozyrakis. 2006. Early Release: Friend or Foe?. In TTM’06. ACM.


