Efficient Correction of Anomalies in Snapshot Isolation Transactions

HEINER LITZ, Stanford University
RICARDO J. DIAS, NOVA-LINCS, NOVA University of Lisbon
DAVID R. CHERITON, Stanford University

Transactional memory systems providing snapshot isolation enable concurrent access to shared data without incurring aborts on read-write conflicts. Reducing aborts is extremely relevant as it leads to higher concurrency, greater performance, and better predictability. Unfortunately, snapshot isolation does not provide serializability as it allows certain anomalies that can lead to subtle consistency violations. While some mechanisms have been proposed to verify the correctness of a program utilizing snapshot isolation transactions, it remains difficult to repair incorrect applications. To reduce the programmer's burden in this case, we present a technique based on dynamic code and graph dependency analysis that automatically corrects existing snapshot isolation anomalies in transactional memory programs. Our evaluation shows that corrected applications retain the performance benefits characteristic of snapshot isolation over conventional transactional memory systems.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

General Terms: Experimentation, Algorithms, Performance

Additional Key Words and Phrases: Transactional memory, snapshot isolation, concurrency, consistency

ACM Reference Format:
Heiner Litz, Ricardo J. Dias, and David R. Cheriton. 2014. Efficient correction of anomalies in snapshot isolation transactions. ACM Trans. Architec. Code Optim. 11, 4, Article 65 (December 2014), 24 pages.
DOI: http://dx.doi.org/10.1145/2693260

1. INTRODUCTION

Transactional memory (TM) improves performance and scalability over locking techniques by increasing concurrency. In the case of locks, concurrency is limited by the size of the critical regions and contention. In TM, concurrency is restricted by the number of data conflicts. As a consequence, techniques that reduce the probability for conflicts are key for improving system performance.

Existing TM algorithms [Herlihy and Moss 1993; Rajwar and Goodman 2002; Hammond et al. 2004; Rajwar et al. 2005; Ananian et al. 2005; Moore et al. 2006; Chung et al. 2006; Bobba et al. 2008; Tomić et al. 2009; Yoo et al. 2013] follow a conservative approach, based on two-phase locking (2PL) for detecting conflicts. The approaches are conservative in that they abort transactions on all read-write conflicts even if committing the transaction would lead to correct behavior. To address this problem, Ramadan et al. [2008], Aydonat and Abdelrahman [2008], Ramadan et al.

Author's addresses: H. Litz and D. R. Cheriton, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, CA 94305; email: heiner.litz@stanford.edu, cheriton@cs.stanford.edu; R. J. Dias, Informatics Department, FCT–NOVA University of Lisbon, 2829-516 Caparica, Portugal; email: ricardo.dias@fct.unl.pt.

DOI: http://dx.doi.org/10.1145/2693260

[2009], and Qian et al. [2014] have proposed supporting conflict serializability (CS), which avoids some of these noncritical conflicts, reducing transaction aborts. CS systems track additional transaction dependencies to check whether a committing transaction violates the global consistency in the case it conflicts with other concurrent transactions.

Snapshot Isolation (SI) [Berenson et al. 1995] has been shown to even further reduce aborts over CS [Litz et al. 2014], in particular for applications that utilize read-only or read-heavy transactions. SI avoids read-write conflicts entirely by taking a snapshot of the system state at the start of the transaction and serving all reads from this snapshot, isolating the transaction from external writes. While 2PL and CS implementations are exposed to conflicting writes whenever those are made visible, SI transactions remain isolated from writes even after they are committed to memory. This is particularly useful for long-running read-only transactions that are permitted to commit in the presence of many conflicting update transactions by operating on an older, consistent snapshot. By virtue of these properties, SI is preferred over CS and has been adopted as the de facto default isolation mechanism for transactional database systems, while receiving only little attention by the TM community. Although, as shown by Dias et al. [2011] and Litz et al. [2014], SI-based TM systems (SI-TM) provide considerably higher performance and scalability than both 2PL and CS systems, SI-TM's dismissal can be explained by its susceptibility to consistency anomalies, which can lead to incorrect program behavior. SI algorithms allow two kinds of consistency anomalies: the *write-skew* and the *SI read-only* anomalies. Hereafter, we will refer to both anomalies as *SI anomalies*.

For instance, a write-skew occurs if two transactions have disjoint write sets, and both transactions concurrently read data items that are modified by the other transaction. This can result in an application constraint violation, leading to incorrect execution [Berenson et al. 1995].

The feasibility of SI in the TM setting greatly depends on the following two issues:

—How to determine whether a program is susceptible to incorrect execution due to si-anomalies
—How to correct a program if it exhibits si-anomalies

The first issue is addressed by previous works such as Dias et al. [2012] and Kuru et al. [2014]. In the former, the authors developed an automatic verification technique to assert whether a program executed under SI may generate si-anomalies during its execution. In the latter, a program annotated with its invariants is statically verified to check whether it is equivalent to a serializable execution when executed under SI.

The second issue is challenging, particularly for large programs, as determining the distinct memory operations that inflict an SI anomaly can be a complex task. We address this challenge in this article by presenting a technique that automatically corrects SI anomalies without programmer intervention and with minimal impact on performance. The technique is based on *promoting* dangerous read operations. Promoted reads [Fekete et al. 2005] are incorporated in the conflict detection phase of the SI-TM implementation, and in the case of SI anomalies, force one of the affected transactions to abort. In a hardware TM (HTM) system, promoted reads are realized by adding a prefix to the `mov` instruction, which adds the address to the write set of the transaction, which in the case of an HTM generally is the L1 cache. In a software TM (STM) system, the compiler instruments the read operation to enforce write set insertion. Note that in both implementations, the entries in the write set do not necessarily correspond with the entries in the write buffer holding speculative writes. To avoid maintaining a separate data structure, a single bit can be used in the write set to distinguish writes from promoted reads.

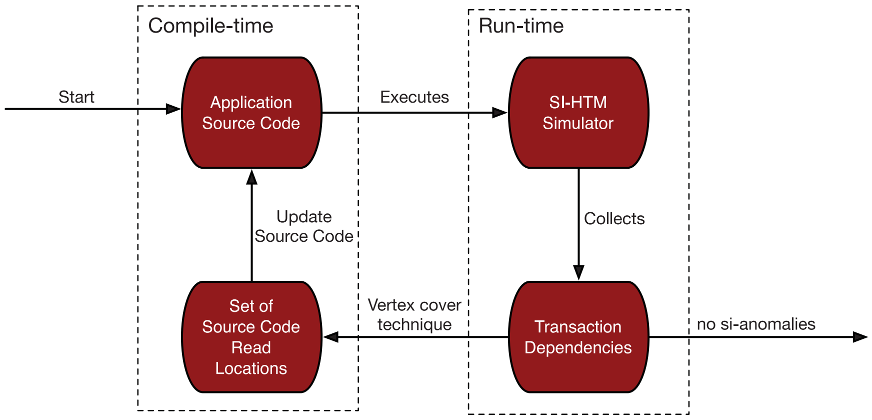

Fig. 1. SI correction tool lifecycle.

Choosing the read operations that should be promoted to eliminate an SI anomaly is known to be an NP-Complete problem from previous work in databases [Jorwekar et al. 2007]. We address this problem by proposing a novel representation of an SI anomaly using a dependency graph focusing on read operations. Additionally, we reduce the problem of choosing the best read operations to be promoted to a graph coverage problem for which there are known polynomial algorithms that can approximate the optimal solution with high accuracy.

To use our correction technique, we need to know the set of read operations that contribute to an SI anomaly. We found that previous approaches on the detection of SI anomalies could not be scaled easily for medium-sized examples, such as the STAMP benchmarks [Cao Minh et al. 2008]. We use execution traces to obtain information about SI anomalies. In this article, we explain how these traces are collected and how we process them to correct the SI anomalies in the application's source code. We focus on SI-based HTMs, although our techniques are equally applicable to STMs. In particular, a program that is corrected with our methodology can be both executed on SI HTMs as well as STMs.

We developed a tool that integrates with the SI-HTM simulator [Litz et al. 2014] to automatically correct the SI anomalies exposed by applications. Figure 1 shows the lifecycle of our tool. In our methodology, an application that exhibits write-skew is first analyzed using PIN [Luk et al. 2005] to collect transaction dependency information. If the obtained dependency information exposes any SI anomaly, we compute a set of source-code read locations that need to be modified using *read promotion* in order to correct the exposed SI anomalies. For each of the computed source-code read locations, the tool automatically modifies the application's source code. The changes made to the application's source code are sufficient to prevent all SI anomalies exposed in the former execution. However, to increase confidence, the corrected application can be analyzed again to find more SI anomalies that were not exposed in previous executions. If no further SI anomalies can be found, the process terminates.

We evaluated our correction technique with a broad range of transactional memory applications. For each we show the impact of the correction on the throughput and abort rates when compared to an uncorrected snapshot isolation version, and with a 2PL serializable TM algorithm. Our results show that, after the correction of SI anomalies, most tested applications retain their high performance characteristic of SI over conventional transactional memory systems.

```
void Withdraw(string account, int value) {
    if (c + s > value)
        if (account == "checking") c = c − value;
        else if (account == "savings") s = s − value;
}
```

Listing 1. Withdraw code exhibiting write-skew.

The contributions of this article are the following:

—A novel representation of SI anomalies using a dependency graph focusing on read operations,
—A set of graph coverage algorithms that minimize the number of read promotions
—A thorough evaluation, with several benchmarks, that exposes the variation of abort rate and throughput according to different promotion strategies
—A description of a tool that can be applied to the source code of arbitrary C/C++ TM applications for removing existing SI anomalies

The remainder of this article is structured as follows. Section 2 provides background information on SI and its consistency anomalies. Section 3 introduces our technique to correct SI anomalies from applications. Section 4 describes optimization techniques to determine the best modifications to be applied to the code. Section 5 presents our tool, while Section 6 evaluates our technique, the different optimization mechanisms, and their effect on the abort rate. Section 7 provides an overview of related work and Section 8 presents conclusions.

2. BACKGROUND

Before introducing our technique, we provide a definition of the SI-based TM model as well as an example of one of the SI anomalies, the write-skew anomaly.

2.1. Snapshot Isolation

Snapshot isolation [Berenson et al. 1995] represents a consistency mechanism that provides higher concurrency than existing TM systems by allowing simultaneous reads and writes to the same data item. SI achieves this property by providing every transaction with a memory snapshot that is taken at transaction begin and stored in a local buffer. All reads are served from the snapshot, thus are isolated from other transactions' writes. Transactions conflict only if they simultaneously write to the same data item, creating a write-write conflict that is handled by aborting the latter of the transactions. SI-based TM systems have been proposed by Riegel et al. [2006], Dias et al. [2011], and Litz et al. [2014]. They use timestamp-based mechanisms to store multiple versions of the same data item, which are then used to provide abort-free read-only transactions. Furthermore, transactions in general ignore read-write conflicts and only abort on write-write conflicts.

SI is appealing for performance reasons; however, it may lead to nonserializable executions, resulting in two kinds of consistency anomalies, which we call SI anomalies: the *write-skew* [Berenson et al. 1995] and the *SI read-only anomaly* [Fekete et al. 2005]. Consider the following example, which suffers from the *write-skew* anomaly. A bank client can withdraw money from a checking and a savings accounts represented by c and s. Executing two instances of the program in Listing 1 can lead to the following transaction history, where we denote $R_T(var, val)$ as a read of variable var that returns the value val by transaction T. $W_T(var, val)$ represents a write operation of transaction T to variable var with value val and C_T the commit of transaction T.

$$H_1 : R_1(c, 20)\ R_2(c, 20)\ R_1(s, 80)\ R_2(s, 80)\ W_1(c, -60)\ C_1\ W_2(s, 0)\ C_2.$$

```
void remove(int value){
  Node prev = head;
  Node next = prev.getNext();
  while(next.getValue()<value){
    prev = next;
    next = prev.getNext();
  }
  if(next.getValue()==value){
    prev.setNext(next.getNext());
    free(next);
  }
}
```

Listing 2. Linked list code exhibiting write-skew.

Executing the history H_1 under SI leads to the final sum of both accounts of -60, a negative value that contradicts the invariant of $c + s > 0$. This execution would be impossible in a serializable TM, as transaction 2 would have to abort due to a read-write conflict on c.

Another, more complex, example of write-skew is given in Listing 2. Consider the following linked list with four elements: $A \mapsto B \mapsto C \mapsto D$. The `remove(B)` operation will transform the list into $A \mapsto C \mapsto D$, while the `remove(C)` operation transforms the list into $A \mapsto B \mapsto D$. Executing both operations concurrently does not induce a write-write conflict; therefore, both operations will be committed, leaving the list in an inconsistent state in which A points to a nonexisting element, dropping D from the list.

2.2. Dependency Graph Analysis

Fekete et al. [2005] proposed to utilize dependency graph analysis to resolve SI anomalies (both the write-skew and the SI read-only anomalies) in transactional database systems running under SI. Each transaction T is defined by its set of read and write accesses $T = \{r_0(x_0), \ldots, r_n(x_n), w_0(y_0), \ldots, w_m(y_m)\}$. Two kinds of read-write dependencies exist between pairs of transactions: two transactions T_0 and T_1 exhibit a write-read dependency $T_0 \xrightarrow{wr-x} T_1$, if T_1 reads a variable $r(x)$ that was written $w(x)$ by T_0; and two transactions T_0 and T_1 exhibit a read-write dependency $T_0 \xrightarrow{rw-x} T_1$, if T_0 reads a variable $r(x)$ that is concurrently written $w(x)$ by T_1. A *dynamic* dependency graph is then defined as an ordered pair $D = (V, A)$ with

—V, the vertex set of all transactions generated by the execution of a TM program
—A, a set of ordered triples (T_0, x, T_1) where T_0 and T_1 exhibit a dependency

$$T_0, T_1 \in V \ \mid \ T_0 \xrightarrow{wr-x} T_1 \vee T_0 \xrightarrow{rw-x} T_1$$

Fekete et al. [2005] showed that the necessary condition for an SI anomaly is a dependency cycle in the dependency graph called *dangerous structure*, which has the same properties for both write-skew and SI read-only anomalies. By detecting these cycles and aborting transactions accordingly, an SI-based system provides serializable semantics. The mechanism is based on dynamic dependency graphs that are constructed by analyzing in-flight transactions at runtime. The challenge of dynamic anomaly detection is that a transaction needs to check for dependency cycles at commit time against all other concurrent transactions even against those that have been committed already. Especially for long-running transactions, this leads to state explosion as an unbounded history of already committed transactions needs to be stored, rendering it impractical for a hardware implementation.

This overhead problem led to the development of techniques based on static analysis to detect SI anomalies in TM programs, as in Dias et al. [2012], and Kuru et al. [2014]. These techniques, however, only allowed *detection* of SI anomalies and did not address the problem of *correcting* them. In this article, we start where previous works stopped, by focusing on the correction of SI anomalies by making changes to the source code of TM programs.

3. SI ANOMALY CORRECTION

In this section, we present a technique to correct existing SI anomalies from TM programs. In order to prove our concept, we need to collect information about SI anomalies in several applications, such as traditional TM benchmarks. Previous proposed approaches to detect SI anomalies in TM programs are limited to small examples, and in the particular case of Kuru et al. [2014] they do not provide the necessary information to correct an SI anomaly.

Due to these limitations, we choose a pragmatic approach based on execution traces of TM applications running under snapshot isolation and analyze those traces to find SI anomalies. This technique, as all techniques based on dynamic execution traces, does not guarantee that we can find all SI anomalies that could be triggered by the application, but we can detect all SI anomalies triggered in a particular run of the application. From our experiments, as shown in Section 6, we obtain a sufficient number of SI anomalies to evaluate our correction technique.

As a first step, applications are profiled to generate the so-called *transaction traces*. These traces are then postprocessed to extract all dependencies between transactions and detect the SI anomalies. We detect both the write-skew and SI read-only anomalies by detecting dangerous structures in the generated dependency graph. Then, optimization techniques are applied to determine the particular source code modifications that resolve the detected SI anomalies with minimal impact on performance. Finally, a script automatically applies the required modifications to the application's source code.

3.1. Trace Generation

Two conditions are necessary for SI anomalies to occur. The affected transactions need to temporally overlap and the transactional read and write operations need to access the same data items. This information can be represented by a *transaction trace,* as shown in Figure 2 using the following notation.

`StartTRX<ID, TS>`, and `End<ID, TS>` denote transaction delimiters with their corresponding globally unique timestamp TS and a unique transaction ID assigned to each transaction defined in the source code. Instances of the same transaction thus use the same ID but can be disambiguated by their timestamps. `Write<ADDR>` denotes a transactional write operation targeting address `ADDR`, while `Read<ID, ADDR>` denote transactional read operations, where `ID` represents a unique identifier assigned to each read operation listed in the source code. `ID` can be, therefore, expressed as a concatenation of source code file and line number in that file.

Notation 3.1. Hereafter, we denote as *read location* the source code location of a call to a transactional memory read function (compile-time property). And as *read operation*, the execution of a TM read function (runtime property). A single read location may generate many distinct *read operations* during the execution.

The trace depicted in Figure 2 includes the following information: (1) start and end timestamps that define temporal overlap of transactions; (2) memory writes; (3) memory reads, colored depending on their `ID`; and (4) read-write dependencies, symbolized by arrows, in consequence to shared data access. Note that a trace may contain multiple instances of the same transactions, as in the case of the two transactions

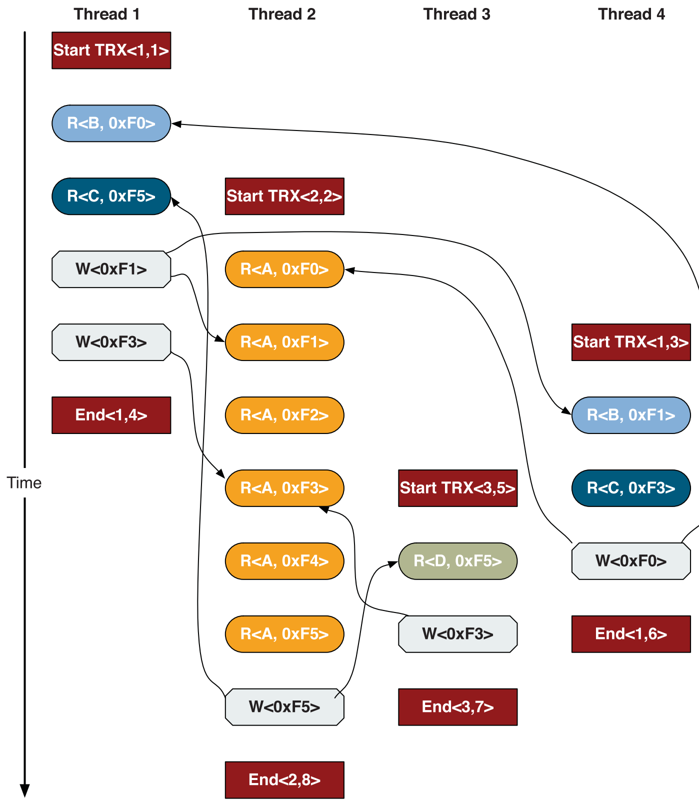

Fig. 2. Example of a transaction trace.

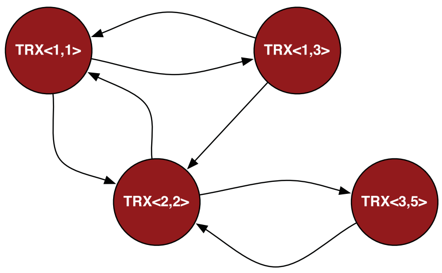

Fig. 3. Dynamic dependency graph.

started at TS = 1 and TS = 3. Furthermore, a read operation with a certain ID may occur multiple times within a single transaction if it is repeatedly invoked, as in a loop.

3.2. Dynamic Dependency Graph

The technique of Fekete et al. [2005] can be used to transform a transaction trace into a dynamic dependency graph (DDG) to enable the detection of SI anomalies. As an example, Figure 3 shows the dynamic dependency graph for the previously shown transaction trace. It shows four dependency cycles, thus four SI anomalies: (1) between TRX<1,1> and TRX<2,2>; (2) between TRX<1,1> and TRX<1,3>; (3) between TRX<1,1> , TRX<1,3>, and TRX<2,2>; and (4) between TRX<2,2> and

TRX<3,5>. DDGs are useful for detecting SI anomalies and have been deployed in transactional database systems for eliminating SI anomalies at runtime. However, as graph analysis introduces significant overheads to TM systems, we propose profiling applications only once, determining all read operations that participate in a cyclic dependency, and then removing them from the source code entirely. Subsequently, the application can be executed with no runtime overheads.

Eliminating SI anomalies through source code modifications requires determination of the memory read operations that induce cyclic dependencies. DDGs enable detection of SI anomalies, however, they do not identify the memory operations that cause the dependencies. Furthermore, while DDGs denote only a single edge between two transactions even if there exists more than a single dependency, our approach needs to determine all possible memory operations that can lead to a dependency as it resolves the SI anomalies at compile time instead of by aborting transactions at runtime. Finally, the size of a DDG is unbounded, growing linearly with the execution time, exacerbating its analysis overhead. To address these issues, we introduce the notion of read dependency graphs (RDGs).

3.3. Read Dependency Graph

Our technique applies read promotion [Fekete et al. 2005] to enable correct program execution. Given the set of all SI anomalies in an application, serializable execution of snapshot transactions can be enforced by promoting the read operations that form dependency cycles. A promoted read conflicts with a write operation that targets the same memory address, leading to the abort of one of the transactions. Promoted reads neither conflict with other promoted reads nor with conventional read operations. By applying promotion, we observe the following: A cyclic dependency involves two or more read operations and promoting only one of them is sufficient to avoid the SI anomaly. Furthermore, memory read operations can be part of multiple SI anomalies and promoting reads might have varying implications on performance. As an example, consider the linked list remove operation in Listing 2, which iterates over the list elements in line 6. Promoting that read location is likely to affect performance negatively as it is executed more frequently than other read locations. Our objective is to choose the set of read locations to be promoted that minimizes the number of new conflicts while maintaining the performance benefits of SI.

To succeed in this objective, we will represent an SI anomaly, which corresponds to a cyclic dependency in a DDG, as a hyperedge[1] connecting all read locations (vertices of a hypergraph) that are part of the SI anomaly. As a result, we can identify the read locations that participate in multiple SI anomalies by computing the degree of each vertex. This identification procedure is equivalent to a graph coverage problem.

To perform the transformation of the representation of SI anomalies for promoting read locations, we introduce the following definitions. A DDG generated from the execution trace of a program P may contain several cyclic dependencies. Each cyclic dependency corresponds to a single SI anomaly. We denote as $\mathsf{Cycles}(P)$ the set of cyclic dependencies generated from the execution of program P.

Definition 3.2 *(Static SI anomaly read-set).* Given a cyclic dependency $c \in \mathsf{Cycles}(P)$ composed by a set of edges $T_0 \xrightarrow{x} T_1 \xrightarrow{y} \cdots \xrightarrow{z} T_0$, let the static SI anomaly read-set $ARS(c)$ be the set of read locations associated with the read operations that occur in each dependency of the cycle:

$$ARS(c) = \{r_i \mid (\_, r_i(x), \_) \in c\},$$

[1]A hyperedge is a graph edge that can connect more than two vertices.

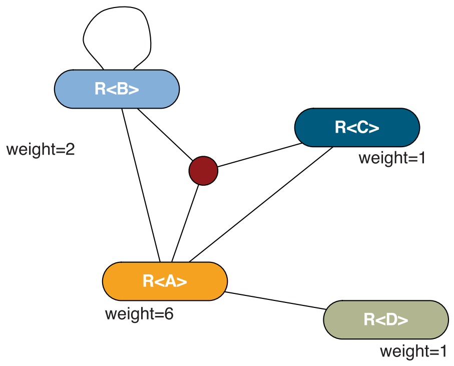

Fig. 4. Read dependency graph.

where the read subscript i represents the unique identifier of the read operation in the source code.

The static SI anomaly read-set corresponds to a sound overapproximation of the concrete read operations that compose the SI anomaly. This overapproximation is achieved by dropping the data item information x from the read operation $r_i(x)$, maintaining only its unique identifier.

LEMMA 3.1 (STATIC SI ANOMALY READ-SET SOUNDNESS). *The static SI anomaly read-set is a sound overapproximation of the concrete set of read operations that compose the respective SI anomaly.*

PROOF. An SI anomaly is composed of a set of concrete read operations $r_i(x)$ where r_i denotes the source code point of such a read operation and x represents the data item accessed. Thus, the set of all distinct read locations r_i without the data item represents all read operations issued at location r_i for any data items. □

Next, we define the RDGs, in which vertices represent read locations, and each edge represents a single SI anomaly.

Definition 3.3 *(Read dependency graph).* Let $G = (R, p)$ be an RDG with $R = r_1, \ldots, r_n$, the set of all read locations defined in the source code of a TM program and p, a set of unordered n-tuples of read locations representing the set of SI anomalies

$$p = \{ARS(c) \mid c \in \text{Cycles}(P)\}$$

The RDG as defined has the following properties:

—It exposes distinct memory read locations that are part of an si-anomaly.
—It expresses multitransaction cyclic dependencies.
—Its size is bounded by the number of read locations defined in the source code.

Figure 4 shows the RDG for the trace in Figure 2. There is one node for each read location that is part of at least one cyclic dependency. Each (hyper)edge of the graph shows one particular SI anomaly. An SI anomaly between two instances of the same transaction is shown in the case of `Read<B>`, which contains an edge to itself. Cyclic dependencies that compose more than two transactions are shown using hyperedges, as in the case of the SI anomaly between `Read<A>`, `Read<B>` , and `Read<C>`. Each node is assigned a weight, indicating the number of times the read was executed during dynamic analysis.

Given a graph G, the problem of finding a valid set of read promotions that eliminate all SI anomalies can be formulated as a vertex cover problem:

Definition 3.4 (*Hypergraph Vertex Cover Set*). A vertex cover of a given hypergraph $G = (V, E)$ is a subset $C \subseteq V$ such that for all edge $e \in E$, there exists a vertex $v \in C$ that is contained in e ($v \in e$).

We can eliminate all SI anomalies captured from the DDG by promoting each read location from the vertex cover set of the RDG. This technique to avoid SI anomalies is captured by the following theorem.

THEOREM 3.2 (SI ANOMALY CORRECTION). *The promotion of all read locations of an RDG's vertex cover set avoids all the SI anomalies identified from the execution trace of a program P.*

PROOF. Each hyperedge of the RDG represents a cyclic dependency on the DDG. Promoting a read location has the effect of removing the read operation from the DDG. Given the set of read locations that are contained in every hyperedge—the cover set—by promoting each of these read locations, the dependency cycles from which the read locations were extracted are broken and thus the SI anomalies are avoided. □

4. CHOOSING A VERTEX COVER

Several vertex cover sets exist for the same graph, therefore given a number of valid vertex covers $C_0, C_1, \ldots, C_m \in \mathsf{CSet}(G)$, where $\mathsf{CSet}(G)$ is the set of all cover sets of graph G, which one should we choose? Every promoted read location participates in conflict detection, thus the selection of C affects both abort rate and performance. In the following, we present three algorithms to determine a valid C.

4.1. Naive Vertex Cover

The most simple and correct solution for the vertex cover problem is to include every vertex of the RDG. Thus, in the example RDG shown in Figure 4, a naive vertex cover is given by $C = \{$`Read<A>`, `Read<B>`, `Read<C>`, `Read<D>`$\}$.

4.2. Approximate Minimum Vertex Cover

The following technique minimizes the number of promoted read operations to limit the read operations that participate in conflict detection.

Definition 4.1 (*Minimal Vertex Cover Set*). Let $|C|$ be the number of vertices in C. A vertex cover $C_0 \in \mathsf{CSet}(G)$ of a given hypergraph $G = (V, E)$ is minimal if $\forall C_i \in \mathsf{CSet}(G) : |C_0| = min(|C_0|, |C_1|, \ldots, |C_m|)$. Such C_0 is called a minimum vertex cover (MVC) of G.

Finding an MVC for an arbitrary G is NP-hard [Garey and Johnson 1979]; however, there are approximation algorithms that execute in polynomial time. We chose to apply a greedy algorithm that picks the vertex with the largest number of edges, removes the vertex and its edges from the graph, and continues until all vertices are covered. Although Papadimitriou and Steiglitz [1998] proved that this heuristic produces vertex covers that can be $\log(|V|)$ times larger than the optimal solution, the greedy algorithm showed close to optimal results for the graphs we studied.

In particular, the obtained results on realistic graphs proved to be superior to the algorithm proposed by Leiserson et al. [2001] based on selecting random edges. Algorithm 1 shows the greedy approximation algorithm. We implemented this algorithm to find an approximate MVC of the RDG within the read promotion step.

Applying Algorithm 1 to the example in Figure 4 leads to a vertex cover of $C_0 =$ {`Read<A>`, `Read<B>`}, which represents an MVC.

ALGORITHM 1: Greedy Approximate MVC Algorithm

```
Input: V : vertex set, E : edge set
Output: C : cover set
C := {};
while $V \neq \emptyset$ do
    determine v ∈ V that has the most edges;
    V := V \ {v};
    C := C ∪ {v};
    forall $e \in E \mid v \in e$ do
        E := E \ {e};
    end
    forall $u \in V$ do
        if $\nexists e \in E \mid u \in e$ then
            V := V \ {u};
        end
    end
end
return $C$;
```

4.3. Weighted Approximate Minimum Vertex Cover

The previous algorithm minimizes the number of promoted read operations. However, different read operations may have varying impact on the abort rate and performance. In particular, read operations as in line 6 of Listing 2 that are executed frequently, if promoted, should expose a higher probability for causing a conflict. To study this effect, for each read location defined in the source code, we count its occurrence in the transaction trace and append this information to each node in the RDG, transforming it into a weighted RDG. With this information, we can compute a vertex cover set that takes into account the occurrence of read operations and minimizes the promotion of read operations with high occurrence. We define a *minimal weight vertex cover set* as follows.

Definition 4.2 (*Minimal Weight Vertex Cover Set*). Given a hypergraph $G = (V, E)$, let W be a set of weights $w(v)$ for each $v \in V$ and $\Sigma_C = \sum_{v \in C} w(v)$. A vertex cover $C_0 \in \mathsf{CSet}(G)$ is minimal if $\forall C_i \in \mathsf{CSet}(G) : \Sigma_{C_0} = min(\Sigma_{C_0}, \Sigma_{C_1}, \ldots, \Sigma_{C_m})$. Such C_0 is called a minimum weighted vertex cover (MWVC) of G.

Finding an MWVC of G is again NP-hard; however, there have been proposed multiple algorithms [Clarkson 1983; Bar-Yehuda and Even 1981; Pitt 1985; Nemhauser and Trotter Jr 1975] that provide approximate solutions in polynomial time. All algorithms provide a vertex cover whose combined weight is less than twice of the optimal solution. In between the four proposals, we chose to implement Clarkson's algorithm [Clarkson 1983] shown in Algorithm 2. It provides the lowest runtime and best approximation ratio, of 1.05 on average [Taoka and Watanabe 2012]. This algorithm requires the notion of vertex degree, which represents the number of edges that contain a specific vertex, and in our case corresponds to the number of cycles indicating potential SI anomalies in which a given source code read location occurs. We define a vertex degree as follows:

Definition 4.3 (*Vertex Degree*). Given a hypergraph $G = (V, E)$, we denote $d(v) = |\{e \mid e \in E \wedge v \in e\}|$ the degree of node v.

ALGORITHM 2: Clarkson's Approximate MWVC Algorithm

```
Input: V : vertex set, E : edge set, $w$ : vertices weights
Output: C : cover set
C := {};
while $V \neq \emptyset$ do
    determine v $\in V$ with $w$(v)/$d$(v) being minimum;
    V := V \ {v};
    C := C $\cup$ {v};
    forall $e \in E \mid v \in e$ do
        E := E \ {e};
        forall $u \in e \mid u \neq v$ do
            $w$(u) := $w$(u) - $w$(v)/$d$(v)
        end
    end
    forall $u \in V$ do
        if $\nexists e \in E \mid u \in e$ then
            V := V \ {u};
        end
    end
end
return $C$
```

In each iteration of the algorithm, we choose the vertex that has the minimum $w(v)/d(v)$ ratio. We require the minimum as we favor vertices with lower weight, that is, read locations with lower occurrence, and simultaneously vertices that have many edges, as in the MVC algorithm, that is, read locations that are part of more SI anomalies.

Applying Clarkson's algorithm to the RDG in Figure 4 leads to the following vertex cover, $C_0 =$ {`Read<C>`, `Read<B>`, `Read<D>`}, which represents an MWVC.

4.4. Discussion: Dependency Generation

Dependency generation can be performed by static or dynamic analysis methods. Our technique relies on dynamic analysis to eliminate false positives and to minimize read promotions. Static techniques, on the other hand, need to be more conservative as they have to trade off precision for soundness and completeness. Conservative dependency information increases the number of dangerous structures, which in return increases the number of reads that need to be promoted, leading to a negative effect on performance. Besides providing less precise transaction dependency information, compiler-based approaches also struggle to obtain information about the execution frequency of a specific read operation, preventing the application of the MWVC algorithm. Nevertheless, static analysis techniques represent a valid alternative whose exploration we leave for future work.

5. SI-ANOMALY DETECTION TOOL

We have implemented our technique as a tool that determines and resolves SI anomalies in TM programs. We discuss the most important and interesting features in this section.

5.1. Trace Generation

Obtaining the `ID` of read operations at runtime, which is required to create fixes for the source code, is challenging. In particular, when intercepting transactional read operations, the originating location in the source file that invoked the operation is unknown. Our first implementation leveraged glibc's `backtrace()` facility, called on each read operation to determine the location of the `TM_READ()` function call in the source code, using stack analysis. While this approach works correctly, it shows the following disadvantages. The call to `backtrace()` introduces significant overheads, which leads to a three-orders-of-magnitude slowdown of the profiled application. This prohibits the analysis of complex applications and distorts their behavior and degree of concurrency. Furthermore, it requires modification of the TM library. Finally, it requires transactional reads and writes to be performed as a function call, which is, as an example, not the case for Intel's TSX [Intel Corporation 2012].

To address these issues, we opted to implement trace generation in PIN [Luk et al. 2005], a dynamic binary instrumentation tool developed by Intel that can be used to inject arbitrary code into application binaries at runtime. PIN provides a rich API to analyze register context information and to instrument individual instructions. Using this API PIN tool, developers define a set of instrumentation routines that the PIN runtime uses to augment the instruction stream at runtime. The PIN runtime instruments the instruction stream only once, then buffers the sequence in a code cache to enable its execution at native processor speed without further overheads. Consequently, lightweight analysis functions can be injected by PIN with minimal effect on the execution characteristics of the observed application. In our case, PIN is utilized to analyze everyTM read and write instruction. Our PIN tool tracks the effective memory addresses of said instructions as well as their instruction pointer (IP). Subsequently, the IP can be provided to the *addr2line* tool of the GNU binary utilities suite to determine the line in the source code that invoked the read operation. To enable stack analysis in the case in which read operations are indirectly called via a chain of function calls, our tool maintains a shadow stack within PIN that tracks the instruction pointer of each `call` instruction, enabling us to perform low overhead stack analysis. This PIN-based technique introduces runtime overheads of below 5x, which leads to acceptable runtime and minimizes the effects on the application behavior. To assess the impact of instrumentation on the behavior of concurrent applications, we compared executions with and without our tool. While the deviation of absolute number of aborts per thread is within 10% for both executions, relative execution time shows the same trends for both the instrumented and uninstrumented executions. In particular, while the absolute overhead of our tool depends on the individual application and the time it spends in transactions, scaling the applications from 1 to 32 threads shows closely matched speedups for both executions.

5.2. SI-Anomaly Detection

To find the existing SI anomalies for each transaction, the tool needs to determine the other transactions it overlaps with and then perform conflict detection between the read and write sets between the transactions. The complexity for validating the read and write sets depends on their average size k. As each element of the read set needs to be compared with each element of the write set for both directions, complexity is $O(2k^2)$. This can be reduced to $O(2klog(k) + 2k)$ by presorting the two write sets after address in $klog(k)$ and then stepping through the two sets in linear time. To determine overlap, the timestamps of each transaction must be compared with all other transactions. Therefore, we sort the trace by merging the presorted per-thread traces in linear time $O(|T|)$ and then compare transaction $t_i \in T$ only to $t_{i+1}, \ldots, t_{i+e}$ where t_{i+e} is the last transaction whose start timestamp is smaller than the end timestamp of t_i. This step has a worst-case complexity of $|T|^2$, if every transaction in the history overlaps with every other transaction. On average, the number of overlapping transactions is much smaller; in particular, transactions of one thread will never overlap with another. In the common case, the average overlap is proportional to the number of threads. The upper bound for the runtime of our tool is thus $O(|T|^2) + O(|T|(2klog(k) + 2k))$.

5.3. Vertex Cover

We implemented both Algorithm 1 and Algorithm 2 to determine a vertex cover of the generated RDG. The runtime of both algorithms is $O(|V|log(|V| + |E|)$ [Clarkson 1983]. The RDG collapses all instances of a read operation into a single vertex; thus its size is proportional to the size of the source code, more specifically the number of instructions in the code segment, of the analyzed application. To achieve coverage for large applications, the tool requires $|T|$ to be large; therefore it can be assumed that $|V| + |E| \ll |T|$. As a result, the SI anomaly detection step determines the runtime of our tool. Nevertheless, it is not feasible to replace the vertex cover heuristics with an optimal scheme, as its runtime gets prohibitive with $|V| > 50$.

5.4. Read Promotion

The vertex cover provided by the tool is utilized to promote a corresponding set of read operations. We therefore provide the address that was obtained in the trace generation step to the *addr2line* tool, which outputs the source file and line number. Subsequently, the transactional read operation in the source code can be exchanged for a promoted read operation. A promoted read is treated by the SI-based TM system as follows. Its address is inserted in the write set and a promoted flag is set for this address. In the case of a write operation to the same address, the flag is cleared. During commit, the entire write set is checked for conflicts. If a promoted read conflicts with a write by another thread, the transaction is aborted. If there are two overlapping transactions that accessed the same datum with promoted reads but no writes, both transactions succeed. After successful validation, a transaction iterates over its write set and commits all writes for which the promoted flag is not set to global memory.

In an SI-based STM, read promotion can be supported by annotating the code with additional instructions to track the addresses of such reads. In an HTM, a `mov` instruction prefix can be used to identify promoted reads. We present an example implementation of such an instruction set architecture (ISA) extension in the evaluation section.

6. EVALUATION

6.1. Methodology

To evaluate our technique, we applied our tool to a broad range of transactional memory applications. The applications are first analyzed and corrected with our tool and then run without any tool instrumentation to do performance measurements. First, we analyze a set of microbenchmarks including a singly and doubly linked list, a binary tree (BTree) and a red-black tree. Then we analyze medium-sized benchmarks including LeeTM [Ansari et al. 2008] and STMBench7 [Guerraoui et al. 2007], as well as a Social Graph and a Pagerank application. Finally, we evaluate seven applications from the STAMP [Cao Minh et al. 2008] benchmark suite. For all STAMP applications, we use the default simulator-sized inputs. STAMP, as well as most of the microbenchmarks, have been obtained from the Rochester Software Transactional Memory (RSTM) [Marathe et al. 2006] framework. All applications have been compiled with GCC 4.7.

As there currently does not exist a processor that implements SI-TM, we perform simulations. The simulated TM system is implemented within the PIN [Luk et al. 2005]-based Zsim [Sanchez and Kozyrakis 2013] simulator utilizing precise functional and timing models. Zsim models an out-of-order multicore processor and is calibrated against Intel Westmere. It simulates a wide range of micro-architectural features including branch prediction, μop decoding, limited instruction issue window width, register renaming, functional unit contention, load store ordering, and reorder buffer, and provides detailed cache and memory models. It has been validated using SPEC CPU2006 and PARSEC [Bienia et al. 2008] and exhibits an average absolute performance error of 11.2%.

We also compare the results with a TM system that models Intel's TSX [Yoo et al. 2013] HTM with eager conflict detection and lazy write buffering. The TSX model has been calibrated against Intel Haswell using the STAMP benchmark suite and shows an average error of below 15% for the transaction abort rate as well as absolute performance. We model infinite read and write sets, excluding the effect of capacity aborts. This simplification represents an advantage for the TM system, as SI-TM does not maintain read sets and the read set is usually larger than the write set.

SI-TM is covered in detail by Litz et al. [2014], therefore, we only summarize the most important features of the system. SI-TM utilizes hardware support for managing multiple data versions in memory and provides an efficient snapshotting mechanism based on immutable data and copy-on-write. SI-TM does not copy memory on transaction initiation to form snapshots but rather compiles them on the fly by intercepting writes such that they create new versions of a data item instead of overwriting data in place. This implements lazy versioning and lazy conflict detection. Lazy validation systems may waste processor cycles by executing a condemned transaction in its entirety, whereas eager systems abort conflicting transactions immediately. The impact of these *zombie* [Dice et al. 2006] transactions can be minimized by performing prevalidation. In prevalidation, writes are validated against the committed state in memory within a transaction, whenever the on-chip network has low bandwidth utilization. This enables reduction of wasteful work while still providing lazy validation semantics. Lazy validation also guarantees forward progress, while eager systems suffer from livelock due to mutual repetitive aborts.

SI-TM does not maintain a read set; however, it tracks writes and promoted reads. To support promoted reads, SI-TM introduces an ISA extension. Particularly, it defines an instruction prefix that can be prepended to `mov` instructions that read from memory. Memory reads exhibiting such a prefix require the hardware to add the instruction's effective address to the hardware-maintained list of promoted reads. To implement these new instructions in the simulated architecture, Zsim provides so called *magic instructions* that can be used to extend the ISA with new capabilities. Conflict detection in SI-TM is not performed via the cache coherency protocol; instead, the write set is sent to the validation module during commit similar to Bulk [Ceze et al. 2006]. The validation module obtains a unique commit timestamp for the transaction and compares the write set, including promoted reads, against the committed state in memory. For any address in the write set, if there is a data item in memory whose commit timestamp is greater than the start timestamp of the transaction, the transaction needs to abort. On commit, the write set is stored to memory by creating a new version tagged with the end timestamp.

The SI anomaly detection tool has been implemented within the simulator to perform SI anomaly analysis transparently to software. If the tool detects an SI anomaly, it proposes a set of read operations for promotion and applies the modifications to the original source code during a post-processing step. We then re-execute the modified applications using an SI-based TM system, comparing it to the uncorrected SI baseline and to the serializable TM system. The main focus of our evaluation is to analyze the effect of SI anomalies and the different correction mechanisms, thus we omit all applications that do not exhibit anomalies from the performance study.

6.2. Correctness

Our proposed technique utilizes dynamic code analysis, therefore, like other race detection techniques, does not provide a proof that all SI anomalies have been removed

Table I. Anomalies of Evaluated Applications

Application	Dangerous Reads	MVC	MWVC	Unmodified	Modified
Linked List	5	1	1	inconsistent	correct
Doubly-Linked List	5	1	1	inconsistent	correct
Red-Black Tree	37	15	24	inconsistent	correct
Queue	0			correct	correct
Heap	0			correct	correct
Array	0			correct	correct
BTree	12	4	4	inconsistent	correct
Bayes[2]	0			correct	correct
Genome	2	1	1	inconsistent	correct
Intruder[2]	0			correct	correct
Kmeans	0			correct	correct
Labyrinth	0			correct	correct
SSCA2	0			correct	correct
Vacation[2]	0			correct	correct
LeeTM	5	5	5	correct	correct
SB7	0			correct	correct
Pagerank	5	2	3	inconsistent	correct
Social Graph	7	3	3	inconsistent	correct

from a program. The quality of the results—in particular, the percentage of detected anomalies—depends on the inputs of the tool and the duration of the analysis. In general, the runtime of the tool needs to increase proportionally with the size of the analyzed code to provide consistent results. To increase the confidence of the results, the tool is executed multiple times; with each iteration of the tool we extend its duration and modify the input parameters. If one execution detects all previously detected anomalies and does not find new anomalies over the previous run, we terminate the analysis. At this point, techniques such as those presented by Kuru et al. [2014] can be applied to verify correctness of the implementation.

Table I shows the output of our tool for the evaluated applications. The *Dangerous Reads* column lists the number of read operations that are part of at least one SI anomaly. Our tool determines a subset of those dangerous reads and promotes them. The *Unmodified* column shows whether the application runs without failure under SI-TM, whereas the *Modified* column shows whether the application behaves correctly after applying our tool. We run multiple test cases and if the application fails for at least one test run we define the execution as failed. Note that we repaired the data structures using our tool before we ran the STAMP applications that utilize those data structures. Hence, anomalies that appear in data structures are not shown for the STAMP benchmarks. It is interesting to note that most detected anomalies appear in data structures, while real-world applications generally exhibit no SI anomalies apart from the data structures that they also use. The explanation is that most applications do not operate directly on primitive data types, but only on data structures.

From a practical standpoint, our technique has been proven successful. As it can be seen in Table I, all applications that exhibit SI anomalies produce inconsistent results running under SI, while repaired programs function correctly for all test runs. The STAMP applications utilizing an uncorrected version of Linked List and Red-Black Tree also fail due to assertions that check for memory consistency; however, they execute correctly if they utilize corrected data structures. We emphasize that achieving

[2]These applications fail when running with uncorrected versions of the list and red-black tree implementations. However, the applications themselves contain no SI anomalies.

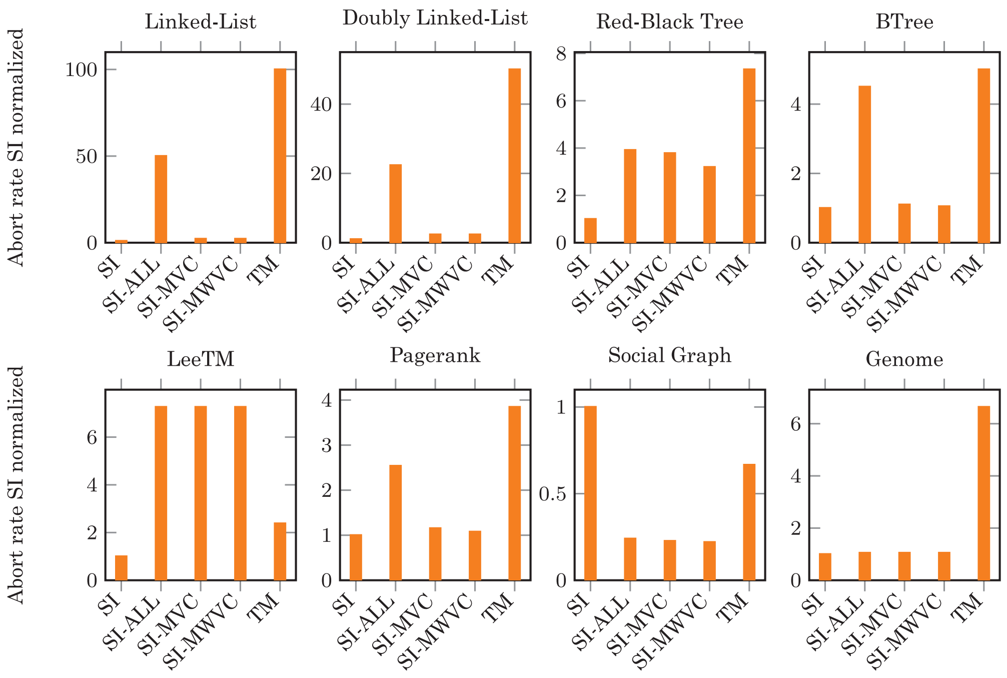

Fig. 5. Abort rates.

absolute correctness is not the main objective of our tool. We regard the proof of correctness as orthogonal, which is addressed by Dias et al. [2012] and Kuru et al. [2014]. Instead, our goal is to provide programmers with a helpful tool that can automatically repair both legacy and newly written applications with minimal performance impact.

6.3. Abort Rate

SI promises to increase performance by reducing aborts to write-write conflicts. However, to address SI anomalies, our technique promotes read operations such that they participate in conflict detection. In theory, an SI-based TM system should outperform a conventional TM system as long as the set of all promoted reads is a proper subset of the read operations defined in the source code. We run all applications that exhibit SI anomalies (see Table I) using the uncorrected SI baseline and compare it to three corrected SI variants plus the TM system. The naive implementation (SI-ALL) promotes every read operation contributing to an SI anomaly. SI-MVC utilizes the method shown in Algorithm 1 to promote reads, whereas SI-MWVC utilizes Algorithm 2.

Figure 5 shows the number of aborts, normalized to SI. We only show applications that exhibit SI anomalies and that require read promotions. All benchmarks have been executed with 32 threads. For Linked-List, Doubly Linked-List and Red-Black Tree we executed 50% `lookup`, 25% `insert`, and 25% `remove` operations on a data structure with 1000 elements. In the case of Linked-List, SI-ALL promotes five read operations in the `insert` and `remove` operations. These promotions cause an increase of 50x more aborts over the SI baseline. `Delete` operations exhibit high abort ratios as SI-ALL promotes the reads of lines `3`, `4`, and `9` in Listing 2. Line `4` appears to be critical as it is called within the loop that is used to traverse the list, although all SI anomalies can be avoided solely by promoting line `9`. SI-MWC and SI-MWVC exploit this fact and generate a vertex cover that delivers much better results. Both reduce aborts by 20x over SI-ALL, having only 2x more aborts than the SI baseline, which shows the importance of promoting the right set of read operations. In the case of the Red-Black Tree, SI-MVC provides no advantages over SI-ALL, whereas SI-MWVC reduces aborts by another 15% over SI-ALL. The benefits of the MVC algorithms are moderate as the three most performance critical read operations form a dependency cycle with themselves. In this case, it is mandatory to promote the read operations for all SI variants.

The binary tree implementation we analyzed supports `lookup`, `insert`, and `delete` operations. It exhibits 12 dangerous read operations, of which only four are promoted by the optimization mechanisms. This has a significant impact on the abort rate, as SI-ALL generates 4x more aborts than the SI baseline, whereas the optimized SI variants only generate between 5% and 8% more aborts than the SI baseline.

The LeeTM benchmark represents a special case in which the number of dangerous reads corresponds to the total number of read locations in the benchmark, and both cover algorithms MWC and MWVC require promotion of all dangerous reads, as in the SI-ALL approach. After the correction, the SI algorithm has no opportunity to ignore the presence of read-write conflicts as all read operations were promoted, and as we can see in Figure 5, the abort rate of SI-ALL, SI-MVC, and SI-MWVC is almost 3x higher than in the TM algorithm. This result can be explained by the fact that, as opposed to the lazy validation of SI, the TM algorithm implements an eager validation strategy that is able to perform better in a workload of high contention, as in the case of the LeeTM benchmark. Another unexpected result of this benchmark was that, although our dynamic analysis detects the presence of SI anomalies, the benchmark always executed correctly under SI without any correction. We further investigated the SI anomalies detected by our dynamic analysis, and found that these detected SI anomalies are not harmful, that is, these SI anomalies do not lead to incorrect results with respect to the benchmark semantics. In the case of SI, which does not promote reads, the SI algorithm reduces the abort rate by more than half compared to the TM algorithm.

The Pagerank application maintains a sparse array of vertices and a sparse adjacency matrix to store undirected edges. We support three operations. The `insert` operation inserts a vertex at a random location within the array if it does not yet exist and inserts a random number of random edges into the adjacency list. As the edges are undirected, each edge insertion requires two writes. The `remove` operation deletes a random vertex and its corresponding edges. `Pagerank` represents a read-only operation that traverses the matrix recursively to determine the vertex with the highest edge count. `Pagerank` is the most costly operation and touches many locations. In SI-ALL, one of the reads in Pagerank is promoted, removing its read-only property, which results in high abort numbers. MWVC promotes three reads instead of two for MVC while reducing aborts by 6% when compared to MVC, which confirms that choosing reads correctly for promotion is important. Both optimized SI variants have only between 7% and 15% more aborts than the SI baseline.

Social Graph represents another graph application that utilizes adjacency lists, one for each vertex, to store its edges. The program supports `insert` and `delete` operations for both vertices and edges as well as a `search` operation based on iterators that, for example, finds all edges that satisfy a condition. Social Graph contains seven dangerous reads, from which three need to be promoted. Unfortunately, one of the performance critical reads that occurs within a loop needs to be promoted in all cases, as the vertex representing this read in the RDG contains an edge to itself. In this case, there exists no alternative vertex that we can promote instead. Note that the uncorrected SI implementation performs worse than all other implementations both in terms of aborts and performance. This can be explained by the fact that an existing SI anomaly causes

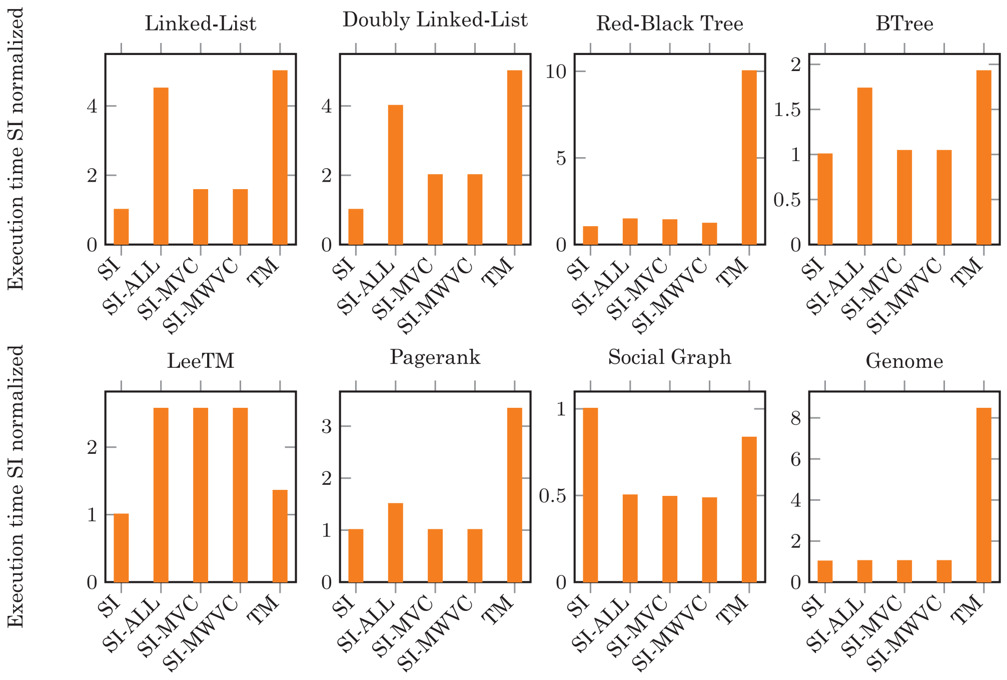

Fig. 6. Speedup—effect of reducing aborts on performance.

the delete operation to fail, which after multiple iterations leads to significantly larger data structures and longer runtimes, as in the correct implementations.

Genome exhibits a write-skew that is conceptually similar to the one shown in the banking example in Listing 1. In particular, to combine two genome segments, it reads their individual sizes and then computes the aggregate size, writing one of the length fields. If two transactions combine the same segments but in a different order, reading the same size fields but writing to distinct size fields, a write-skew occurs. The write-skew can be resolved by promoting one of the two read operations, which has no noticeable impact on the abort rate.

6.4. Performance

The effect of reducing aborts on performance for the different applications is shown in Figure 6. For Linked-List and Doubly LinkedList, SI-ALL shows a performance slowdown of 4x while SI-MWC and SI-MWVC show a slowdown of only 1.5x to 2x when compared to the SI baseline. Still, all corrected SI variants are faster than the TM algorithm, which is 5x slower that the SI baseline. It is interesting to note that the benefits provided by SI over TM increase with both a higher lookup ratio and larger data structures. In the case of larger data structures, lookup operations have a longer duration on average, as they access more elements. This fact increases the cost of aborts for read-only transactions in the case of TM. Both tree implementations benefit substantially from SI. Although the Red-Black Tree exhibits a large number of anomalies, performance is similar between the different promotion techniques.

Again, in the special case of LeeTM, the algorithms SI-ALL, SI-MWC, and SI-MWVC perform worse than the TM algorithm, since they detect all read-write conflicts that may appear during the execution of the benchmark. On the other hand, without the promotion of the dangerous reads, the SI algorithm is faster than the TM algorithm, demonstrating the effectiveness of SI in high contended workloads. Pagerank and Social Graph can both translate reduced abort rates into higher performance. In the specific case of Pagerank, the optimized promotion SI variants have similar execution time as the SI baseline, and are 3x faster than the TM algorithm.

In Genome, read promotion has no noticeable impact on performance for all three promotion strategies. The SI variants show an 8x performance increase over TM.

7. RELATED WORK

In the field of transactional memory, SI has been applied by Riegel et al. [2006], Dias et al. [2011], and Litz et al. [2014]. The detection of SI anomalies in TM programs was first approached by Dias et al. [2012], who propose a static analysis technique to identify possible SI anomalies that can then be manually corrected in the source code. To the best of our knowledge, our work represents the first proposal of automatic correction of SI anomalies in TM programs.

The serialization anomalies introduced by the use of SI are a well studied topic in the database community. In particular, the seminal work of Fekete et al. [2005] laid the foundation for developing systematic approaches to the detection of SI anomalies in database applications. The work presents the theory of SI anomaly detection and a syntactic analysis to detect SI anomalies for the database setting. The proposed analysis is informally described and applied to the database benchmark TPC-C [Transaction Processing Performance Council 2010]. A sequel of that work [Jorwekar et al. 2007] describes a prototype that is able to automatically analyze database applications.

Regarding the correction of SI anomalies, two different lines of research were followed by the community: *offline* correction of anomalies and *online* prevention of anomalies. The *offline* correction techniques resolve SI anomalies at compile time. Therefore, the approaches identify anomalies *a priori* using dynamic or static analysis methods and correct them either by modifying the source code of the application or by executing a subset of the transactions in a stronger isolation level. The work described in this article fits in this category. Two approaches for offline SI-anomaly correction have been presented by Fekete et al. [2005]. The first one is based on the injection of a dummy write operation on a specific data item common to all transactions that participate in the SI anomaly. The second approach is based on the promotion of read operations, as we perform in our work. The promotion of a read operation allows the operation to become part of the transaction validation process, thus it will force a conflict with any write operation on the same data item. In both techniques, the decision of where to inject the dummy writes, or which read operations are promoted, is left to the developer. Our approach improves on those techniques by performing the read promotion automatically. Furthermore, we determine the minimal subset of read operations that need to be promoted.

Jorwekar et al. [2007] define the problem of selecting the subset of transactions that need to be modified in order to remove the existing *dangerous structures*, thus avoiding all SI anomalies. In our work, we present several algorithms to tackle this problem by using a novel read dependency graph representation in which the focus is on read locations rather than on transactions. Alomari et al. [2008] present a comparative study on the performance impact imposed by the different correction techniques. They use a small benchmark to compare the performance overhead imposed by using either the materialization, promotion, or running under serializable approaches. In our work, we evaluate the different correction strategies using micro-, medium-, and macro-sized TM benchmarks. While it is impractical to apply offline techniques to transactional database systems that need to support ad-hoc queries, for TM applications, the set of transactions is usually defined at compile time—although there are exceptions, including self-modifying code and dynamically typed languages.

The *online* prevention of anomalies is characterized by preventing the anomalies from occurring during the execution of the program. This requires tracking of dynamic dependencies between transactions and applying lightweight decision procedures whenever the transactions need to commit. Cahill et al. [2008], Revilak et al. [2011], Jung et al. [2011], and Ports and Grittner [2012] follow this approach and guarantee serializability in centralized, or replicated, databases while using an SI implementation. This approach introduces nonnegligible overheads, which are tolerable in I/O-constrained database systems but less so in latency-sensitive TM systems.

Apart from SI, there are other TM systems that relax consistency to improve performance. DATM [Ramadan et al. 2008], SONTM [Aydonat and Abdelrahman 2010], Wait-n-GoTM [Jafri et al. 2013] as well as OmniOrder [Qian et al. 2014] are systems that implement conflict serializability. These systems check additional dependencies to commit transactions in the presence of certain conflicts. The set of transaction histories supported by CS neither includes, nor is included in, the set of transaction histories of SI [Normann and Østby 2010] and both SI and CS use different notions for dependency [Litz et al. 2014]. In particular, SI never aborts read-only transactions, whereas in CS this is possible. One further advantage of SI is that snapshots are isolated from writes even after the writes have been committed. It has been shown that SI reduces aborts and improve performance over CS [Litz et al. 2014].

8. CONCLUSION

Snapshot Isolation is the default consistency mechanism of all major database systems as it avoids read locks and generally provides the best performance of the established consistency models. Nevertheless, SI has been almost ignored by the TM community. The premise of this work is that programmers will be prepared to deal with a relaxed memory consistency of SI in exchange for higher concurrency and performance if there is a practical means to detect and correct SI anomalies. SI permits the write-skew and SI read-only anomalies, which exacerbates reasoning about application correctness. In this article, we show that it is feasible to automatically detect and correct SI anomalies in a wide variety of applications written to use SI transactions. In particular, we have shown that: (1) dynamic analysis provides a feasible tool to detect the exact locations of SI anomalies in TM programs; (2) choosing the best set of read operations for promotion is important; and (3) automatically corrected SI transactions maintain performance benefits over conventional TM systems. Particularly, we show a 50x reduction in aborts and 4x increase in performance for certain benchmarks.

There is a long history of trading consistency for performance in computer systems. Examples include sequential consistency in distributed systems, contemporary multicore systems that have dropped sequential consistency for multithreaded applications, and recent non-ACID compliant NoSQL databases. The success of such systems depends on the balance of performance benefits and the increase in programming complexity. We believe it is time to reconsider the trade-off between performance and consistency for TM. SI provides significant performance gains for a broad range of applications, while our technique enables programmers to correct applications automatically with minimal performance impact.

REFERENCES

Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The cost of serializability on platforms that use snapshot isolation. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering (ICDE'08)*. IEEE Computer Society, Washington, DC. DOI:http://dx.doi.org/10.1109/ICDE.2008.4497466

C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. 2005. Unbounded transactional memory. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*. IEEE.

Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. 2008. Lee-TM: A non-trivial benchmark for transactional memory. In *Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'08)*. LNCS, Springer.

Utku Aydonat and Tarek Abdelrahman. 2008. Serializability of transactions in software transactional memory. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*.

Utku Aydonat and Tarek S. Abdelrahman. 2010. Hardware support for relaxed concurrency control in transactional memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. IEEE.

Reuven Bar-Yehuda and Shimon Even. 1981. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms* 2, 2, 198–203.

Hal Berenson, Phil Bernstein, Jim N. Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD'95)*.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM Press, New York, NY, 72–81.

Jayaram Bobba, Neelam Goyal, Mark D. Hill, MMichael M. Swift, and David A. Wood. 2008. Tokentm: Efficient execution of large transactions with hardware transactional memory. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society.

Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. ACM, New York, NY, 729–738. DOI:http://dx.doi.org/10.1145/1376616.1376690

Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'08)*.

Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk disambiguation of speculative threads in multiprocessors. *ACM SIGARCH Computer Architecture News* 34, 2, 227–238.

JaeWoong Chung, Chi Cao Minh, Austen McDonald, Travis Skare, Hassan Chafi, Brian D Carlstrom, Christos Kozyrakis, and Kunle Olukotun. 2006. Tradeoffs in transactional memory virtualization. In *ACM SIGARCH Computer Architecture News*, Vol. 34. ACM, New York, NY, 371–381.

Kenneth L. Clarkson. 1983. A modification of the greedy algorithm for vertex cover. *Inform. Process. Lett.* 16, 1, 23–25.

Ricardo J. Dias, Dino Distefano, João C. Seco, and João M. Lourenço. 2012. Verification of snapshot isolation in transactional memory java programs. In *ECOOP 2012—Object-Oriented Programming (Lecture Notes in Computer Science)*, James Noble (Ed.), Vol. 7313. Springer-Verlag, Berlin, 640–664.

Ricardo J. Dias, João M. Lourenço, and Nuno M. Preguiça. 2011. Efficient and correct transactional memory programs combining snapshot isolation and static analysis. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Parallelism (HotPar'11)*. Usenix Association.

Dave Dice, Ori Shalev, and Nir Shavit. 2006. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing (DISC'06)*. Springer-Verlag, Berlin, 194–208. DOI:http://dx.doi.org/10.1007/11864219_14

Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. 2005. Making snapshot isolation serializable. *ACM Trans. Database Syst.* 30, 2, 492–528.

Michael R. Garey and David S. Johnson. 1979. *Computers and intractability*. Vol. 174. Freeman, New York, NY.

Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: A benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS / EuroSys European Conference on Computer Systems 2007 (EuroSys'07)*. ACM, New York, NY, 315–324. DOI:http://dx.doi.org/10.1145/1272996.1273029

Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. 2004. Transactional memory coherence and consistency. In *Proceedings of the 31st AnnualInternational Symposium on Computer Architecture (ISCA'04)*. IEEE Computer Society, Washington, DC, 102–113.

Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*. ACM, New York, NY, 289–300. DOI:http://dx.doi.org/10.1145/165123.165164

Syed Ali Raza Jafri, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2013. Wait-n-GoTM: Improving HTM performance by serializing cyclic dependencies. *ACM SIGARCH Computer Architecture News* 41, 1, 521–534.

Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. 2007. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*. VLDB Endowment, Vienna, Austria, 1263–1274.

Hyungsoo Jung, Hyuck Han, Alan Fekete, and Uwe Röhm. 2011. Serializable snapshot isolation for replicated databases in high-update scenarios. *Proc. VLDB Endow.* 4, 11, 783–794.

Ismail Kuru, Burcu Kulahcioglu Ozkan, Suha Orhun Mutluergil, Serdar Tasiran, Tayfun Elmas, and Ernie Cohen. 2014. Verifying programs under snapshot isolation and similar relaxed consistency models. In *Proceedings of the 9th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT'14)*.

Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, and Thomas H. Cormen. 2001. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Heiner Litz, David Cheriton, John P. Stevenson, Amin Firoozshahian, and Omid Azizi. 2014. SI-TM: Reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. PIN: Building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, Vol. 40. ACM Press, New York, NY, 190–200.

Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. 2006. Lowering the overhead of nonblocking software transactional memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*.

Intel Corporation. 2012. Chapter 8: Intel transactional synchronization extensions. In *Intel Architecture Instruction Set Extensions Programming Reference*.

Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. 2006. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*. IEEE Computer Society, Austin, TX.

George L. Nemhauser and Leslie E. Trotter Jr. 1975. Vertex packings: Structural properties and algorithms. *Mathematical Programming* 8, 1, 232–248.

Ragnar Normann and Lene T. Østby. 2010. A theoretical study of 'Snapshot Isolation'. In *Proceedings of the 13th International Conference on Database Theory*. ACM Press, New York, NY, 44–49.

Christos H. Papadimitriou and Kenneth Steiglitz. 1998. *Combinatorial optimization: algorithms and complexity*. Courier Dover Publications, North Chelmsford, MA.

Leonard Brian Pitt. 1985. *A simple probabilistic approximation algorithm for vertex cover*. Yale University, Department of Computer Science, New Haven, CT.

Dan R. K. Ports and Kevin Grittner. 2012. Serializable snapshot isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12, 1850–1861. http://dl.acm.org/citation.cfm?id=2367502.2367523.

Xuehai Qian, Benjamin Sahelices, and Josep Torrellas. 2014. OmniOrder: Directory-based conflict serialization of transactions. In *Proceedings of the 41st Annual International Symposium in Computer Architecture (ISCA'14)*.

Ravi Rajwar and James R. Goodman. 2002. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. IEEE Computer Society.

Ravi Rajwar, Maurice Herlihy, and Konrad Lai. 2005. Virtualizing transactional memory. In *Proceedings. 32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE.

Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. 2008. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*. IEEE Computer Society, Washington, DC, 246–257. DOI:http://dx.doi.org/10.1109/MICRO.2008.4771795

Hany E. Ramadan, Indrajit Roy, Maurice Herlihy, and Emmett Witchel. 2009. Committing conflicting transactions in an STM. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'09)*. ACM, New York, NY, 163–172. DOI:http://dx.doi.org/10.1145/1504176.1504201

Stephen Revilak, Patrick E. O'Neil, and Elizabeth J. O'Neil. 2011. Precisely serializable snapshot isolation (PSSI). In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'11)*. 482–493.

Torvald Riegel, Christof Fetzer, and Pascal Felber. 2006. Snapshot isolation for software transactional memory. In *Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*.

Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium in Computer Architecture (ISCA'13)*.

Satoshi Taoka and Toshimasa Watanabe. 2012. Performance comparison of approximation algorithms for the minimum weight vertex cover problem. In *Proceedings of the 2012 IEEE International Symposium on Circuits and Systems (ISCAS'12)*. *IEEE*, 632–635.

S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. 2009. EazyHTM: Eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM Press, New York, NY, 145–155.

Transaction Processing Performance Council. 2010. TPC-C Benchmark, Revision 5.11.

Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of intel&Reg; transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*. ACM, New York, NY, Article 19, 11 pages. DOI:http://dx.doi.org/10.1145/2503210.2503232

Received May 2014; revised November 2014; accepted November 2014