2012 IEEE 26th International Parallel and Distributed Processing Symposium

SUV: A Novel Single-Update Version-Management Scheme for Hardware Transactional Memory Systems

Zhichao Yan*, Hong Jiang†, Dan Feng*✉, Lei Tian*† and Yujuan Tan*
*School of Computer Science
Wuhan National Laboratory for Optoelectronics
Huazhong University of Science and Technology, Wuhan, China
Email: zhichao_yan@smail.hust.edu.cn, {dfeng,ltian}@mail.hust.edu.cn, tanyujuan@gmail.com
†Department of Computer Science & Engineering
University of Nebraska-Lincoln, Lincoln, USA
Email: jiang@cse.unl.edu

***Abstract*—In order to maintain the transactional semantics, Transactional Memory (TM) must guarantee isolated read and write operations in each transaction, meaning that it must spend a non-negligible and potentially significant amount of time on keeping track of the transactional modifications in its undo or redo log and switching to the proper version at the end of each transaction. Existing TMs failed to minimize the overheads incurred by these operations that are poised to impose more significant TM overheads in current and future many-core CMPs. A direct consequence of this is that extra and different data movements are needed to manage these modifications depending on commit or abort. To address this problem, we propose a novel Single-Update Version-management (SUV) scheme to redirect each transactional store operation to another memory address, track the mapping information between the original and redirected addresses, and switch to the proper version of data upon the transaction's commit or abort. There is only one data update (movement) in our SUV regardless of commit or abort, thus significantly reducing the TM overheads while allowing it to exploit more thread parallelism. We use SUV to replace version-management schemes in some existing hardware TMs to assess SUV's performance advantages. Our extensive execution-driven experiments show that SUV-TM consistently outperforms the state-of-the-art HTM schemes LogTM-SE, FasTM and DynTM under the STAMP benchmark suite. Moreover, we use CACTI to estimate the hardware overheads of SUV and find it is feasible in hardware implementation.**

I. Introduction

Synchronization primitives with various complexities and overheads have been widely used to ensure the correct execution of multiple threads competing for the shared resources in the increasingly common multicore- and manycore-based multiprocessing environment. There have been a plethora of studies on the tradeoff between performance and programmability of various synchronization primitives. Among existing proposals, Transactional Memory (TM) [1] is arguably one of the most competitive methods to hide synchronization complexity while providing good performance. Apart from a great deal of attention being paid by the academic community, the TM technology has been employed in commercial products by a number of leading industrial companies in their processor chips, including Sun Microsystems [2], Azul [3] and IBM. In particular, IBM's Blue Gene/Q [4], a general-purpose commercial microprocessor that employs this technology, will be shipped in 2012 to be used as the building blocks for the Sequoia supercomputer at Lawrence Livermore National Labs [5], which represents a big leap forward for the TM technology in its applications in the real world.

A TM system organizes the potentially racing blocks' executions in transactions, which are each composed of a series of program instructions and are ensured to execute atomically and in isolation. To maintain atomicity and isolation, two key designs, namely, *conflict management* and *version management*, are employed to determine when to detect and how to resolve a conflict, and where to store the speculative modifications and how to merge the uncommitted data with the safe memory respectively. And they must be carefully designed to appropriately complement and traded off between them to achieve the best performance/cost ratio [6].

The transactional notion guarantees that the transaction's modifications either perform fully on commit or none at all on abort, and any modifications during the transaction's execution cannot be accepted until the transaction has committed its work successfully. In particular, a transactional read or write operation must acquire the access permission and hold it until the end of the transaction, an important TM property known as *isolated-read* or *isolated-write*. Any memory access violating the isolated-read or isolated-write property is regarded as a *transactional conflict* and needs a proper solution to maintain the isolation. Time spent on holding the permission is called the *isolated-read window* or *isolated-write window*, which determines the *exclusive period* for each shared variable, a key factor of contention. The time spent on managing the transactional modifications by the version management constitutes a significant part of the *isolation window* (isolated-read window or isolated-write

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.22

Table I: The Abort Behaviors Reported in Published Studies

Study	Abort Ratio	Evaluation Environment and Workloads
LogTM [7]	up to 15%	Splash2 applications run under LogTM
PTM [8]	up to 24%	Splash2 applications run under PTM
LogTM-SE [9]	30% to 40%	Raytrace and BerleleyDB aborted about 30% and 40% transactions respectively
FasTM [10]	up to 4.0%	Evaluate Micro-benchmarks, Splash2 and STAMP under FasTM
SBCR-HTM [11]	up to 75.9%	Evaluate STAMP under HTM with Speculation-Based Conflict Resolution
LiteTM [12]	up to 79.4%	Evaluate STAMP under TokenTM
Lee-TM [13]	up to 72%	Five implementations of Lee's routing algorithm run under DSTM2
TransPlant [14]	up to 79%	Automatically generated a program based on the desired characteristics
RMS-TM [15]	up to 69%	Selected RMS applications run under Intel's prototype STM compiler

window for each shared variable). It is therefore critical to reduce this time by optimizing the version management, the focus of this paper, in order to minimize the forced serialization among multiple transactions, which in turn enables more transactional parallelism to be exposed, especially under high-contention workloads.

In general, there are two main categories of version-management schemes, the optimistic schemes (also known as the eager schemes) and the pessimistic schemes (also known as the lazy schemes). Optimistic version-management is optimized for commits while pessimistic version-management is optimized for aborts because the former is premised on the unlikelihood of ever aborting a transaction while the latter incurs more overheads on commit than on abort. However, we argue that neither is efficient for many-core CMPs of the future generation where there will be increasingly more memory conflicts that must be resolved with a lot of coarse-grained transactions. This is because, while the former must maintain the undo log upon each transactional store to hold the old data and require extra data movement on abort, the latter must maintain the redo log to hold the new data and redo all the transactional store operations on commit and its buffer overflow will result in additional memory accesses. Optimistic schemes are designed to optimize commits that are generally considered more likely than aborts, for which pessimistic schemes are ill-suited. However, recent observations of increasing abort ratios in coarse-grained and high-contention applications demand that overheads due to aborts must also be reduced in order to expose more parallelism that would otherwise be blocked by lengthy and/or frequent aborts. As implied by published results on the significant abort ratios of representative applications listed in Table I, we believe both commit and abort operations should be optimized, especially in the future many-core environment where thread contentions are expected to be very high. This issue provides an opportunity to optimize existing transactional memory systems so that they will perform consistently well on both commit and abort operations, which is one of the main motivations for this paper.

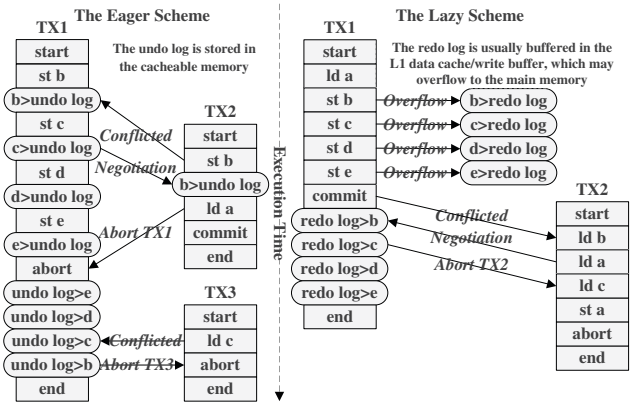


Figure 1: Repair and Merge Pathologies in Existing Version Management Schemes Lengthen Their Isolation Windows

Our preliminary study of the existing version-management schemes based on the high-contention and coarse-grained workloads, such as the STAMP benchmark suite, reveals two main sources of TM overheads [16]. One source of TM overheads is the data movements involving the undo log used in the optimistic schemes such as LogTM [7] that can induce the repair pathology on abort, in which the extra time spent on abort to repair the old data will lengthen the isolation window of the aborted transaction's shared data while attempted accesses to these shared data by the surrounding transactions will lead to transactional conflicts that will block the thread level parallelism. This pathological case will become more pronounced under the high-contention and coarse-grained workloads and possibly lead to a vicious cycle. Another source of TM overheads is the data movements resulting from the data overflow of the redo log used in the pessimistic schemes such as TCC [17] that can induce the merge pathology on commit, in which the extra time spent on commit to merge the new data with the memory will lengthen the isolation window of the committed transaction's shared data while the attempted accesses by the surrounding transactions to these shared data will lead to transactional conflicts that will block the thread level parallelism. Again, this case will also worsen under the high-contention and coarse-grained workloads to induce a possible vicious cycle. Even the hybrid schemes that combine elements of the optimistic and pessimistic approaches in some way and can partially mitigate the repair pathology, such as FasTM [10], are still vulnerable to the repair and merge pathologies. Both the repair and merge pathologies stem from extra data movements in the version-management schemes that lengthen the isolation-window to introduce more transactional conflicts with the surrounding transactions, as elaborated next.

To better understand the problem, we use an example to illustrate the repair and merge pathologies respectively in Figure 1. In an optimistic scheme, apart from the repair time on abort to restore the pre-transaction old state, extra time is needed to store the old value to the undo log before updating the new value in-place upon each transactional modification.

Both may lengthen the isolation-window for the shared variables especially on abort where the surrounding transactions' accesses to these shared variables will conflict with the aborting transaction during its restoration process. In this example, $TX3$ is aborted due to the conflicted access to the shared data during $TX1$'s restoration process. And had $TX1$'s abort operation taken less time, this conflict would have been avoided, thus exposing more thread level parallelism. Fortunately, since optimistic schemes such as LogTM couple the undo-log to hold the old data and update new data in-place, they are prevented from the data overflow from the limited buffer that is unavoidable in pessimistic schemes such as TCC under the coarse-grained workloads. While pessimistic schemes can hide the data movement latency by organizing the new data in the redo-log and storing it to the L1 cache or write buffer with the aid of the improved cache coherence protocol, this data can still be overflowed to the main memory due to the capacity and conflict problems, which will degrade application's performance significantly. For the example in Figure 1, if $TX1$'s commit operation had spent less time on merging the overflowed new data in the redo log, $TX2$ would not have conflicted with $TX1$, thus exposing more thread level parallelism. However, as more and more coarse-grained and high-contention workloads are introduced in TMs, the adverse impact of the repair and merge pathologies will likely become one of the main forces preventing the thread level parallelism from being exposed and exploited, especially in future TM applications of coarser granularity and higher contention.

To overcome these shortcomings of existing version-management schemes in hardware transactional memory (HTM), we propose a novel *Single-Update Version-management* (*SUV*) approach for HTM, called *SUV-TM*, which has the potential to significantly outperform the state-of-the-art version-management schemes. The basic idea behind our SUV is to redirect each transactional store to another memory address, while tracking the mapping between the original and redirected addresses, and switching to the proper version of data upon commit or abort. The salient feature of our SUV-TM approach lies in the fact that only *one* update operation is needed whether the transaction commits or aborts, thus avoiding the overheads due to extra data movements. More importantly, these eliminated overheads imply narrowed isolation windows (i.e., forced serialization of transactions) for HTMs, thus unlocking more transactions earlier and exposing more thread parallelism for performance gains.

Through the work of this paper, we aim to make the following contributions:

(1) We propose a novel Single-Update Version-management (SUV) scheme for HTM to overcome the shortcomings of the existing state-of-the-art version-management schemes that incur significant overheads on both commit and abort;

(2) We prototype SUV-TM in a popular CMP simulator and conduct extensive execution-driven evaluations in both fine-grained and coarse-grained transactional applications, and the experimental results show that SUV-TM outperforms the state-of-the-art LogTM-SE and FasTM schemes by 56% and 9% respectively for all applications of the STAMP benchmark suite, and 95% and 12% respectively for the five high-contention applications of the STAMP benchmark suite. Further, the experimental results show that the DynTM integrated with SUV as its version-management scheme outperforms the original DynTM based on FasTM by 9.8% and 18.6% respectively for all applications and the five high-contention applications of the STAMP benchmark suite.

(3) We use CACTI to estimate the overheads of SUV, especially on access time, energy consumption and area of the first level fully-associative redirect table (the main component used in SUV), and the results show that SUV is feasible in hardware implementation.

The rest of the paper is organized as follows. Section II introduces the necessary background and related work. Section III describes the architecture of SUV-TM. Design and implementation are presented in Section IV. The evaluation environment and experimental results are presented and analyzed in Section V. The paper is concluded in Section VI.

II. Background and Related Work

It has been a long-standing problem to find more efficient alternatives to the notorious lock in shared memory systems to exploit more thread parallelism and make parallel programming easier [1]. Existing solutions have performance benefits but are too hard for ordinary users to write correct programs. The main drawback of these methods lies in their lack of efficient mechanisms to manage complexity. Emerging as a new programming model, TM provides the ability of composition to ease the burden on programmers to write parallel programs correctly and enhances the abstraction level by using transactions to organize the parallel executions.

Since version management in TM is the main thrust of this paper, we will concentrate on introducing the background and existing research in the literature related to this issue. Most research on TM in the literature focuses on space virtualization to support the overflowed transactions, instead of optimizing version management. We believe that the reasons for this focus are twofold. First, earlier transactional workloads are largely transformed from highly optimized lock-based programs, resulting in transactions in these workloads being generally fine-grained and low-contention. The fine-grained and low-contention nature of these workloads makes the overhead of version management relatively low. Second, since most side effects of the TM execution can be attributed to the conflicting accesses from the surrounding transactions,

whose behaviors are too complex to be effectively handled by the existing conflict management policies, a great deal of effort has been put to designing novel conflict management schemes while ignoring the important impact of version management.

In this paper, we argue that version management in TM has a significant impact on TM overhead and it is critical to reduce the isolation window by optimizing version management to minimize the forced serialization among multiple transactions, thus exposing and exploiting more transactional parallelism, especially under high-contention workloads. Unfortunately, existing version-management schemes by and large fail to minimize the isolation window [18]. For example, existing optimistic version-management schemes such as LogTM [7], OneTM [19], LogTM-SE [9] and TokenTM [20] must write the old values to the undo log in a thread's private space before updating the new values in place and trapping into a software library to restore the old values on abort. This leads to one load and one store on commit, along with an extra load and store on abort for each transactional write. In particular, trapping into a software library on abort under high-contention applications dramatically widens the isolation window, which will in turn induce more conflicts, leading to a vicious cycle and thus significantly degrading the performance (See Figure 1). Meanwhile, there are at least two extra cache accesses that result from reading the old value and writing it to the undo log in order to maintain the undo records. Ideally, the cost of these extra accesses should be hidden by subsequent non-memory instructions, thereby avoiding processor stalls. However, it may be difficult to hide these accesses without adding extra L1 data ports as the number of instructions per cycle increases or facing a stream of continuous transactional store instructions [21]. On the other hand, pessimistic version-management schemes such as TCC [17], LTM [22], VTM [23], RTM [24], Scalable-TCC [25], FlexTM [26] and Rock processor [27] buffer the speculative values in a private memory space and broadcast the new values on commit, leading to added time on commit that may in turn lengthen the isolation windows. Additionally, the limited private memory space can easily *overflow* under coarse-grained transactions, incurring extra memory accesses and further degrading the performance. For example, when a block is evicted from the buffer, additional time is needed to fetch the data from the main memory. To make things worse, the amount of data moved along this route can be very significant if the miss happens in a loop structure. UTM [22], PTM [8] and XTM [28] are proposed to handle the overflowed transactions. Meanwhile, Object-Aware HTM [29] is proposed to exploit the data organization of object-oriented language, which updates an object's pointer to point to the new data on commit and requires a significant change to the existing memory management system to organize and manage data in objects. FasTM [10] exploits the inconsistency between the L1 cache and the higher memory hierarchy to prevent the new value from spreading to the higher memory hierarchy until the transaction commits its work. However, whenever overflow occurs, it degenerates to LogTM-SE. Recently, DynTM [30], based on FasTM [10], is proposed to combine the advantages and avoid the disadvantages of existing version-management and conflict-management schemes, but it is highly dependent on the history-based selector (i.e., a predictor) to choose the execution mode of the transactions and degenerates to the existing schemes with inaccurate predictions, thus again suffering from the known side effects of data movements. Concurrently with our SUV scheme, A. Armejach et al. [31] propose to use a reconfigurable L1 cache to reduce the data movement overheads, which requires more silicon area for the reconfigurable cache that is dedicated for transactional access operations exclusively.

Generally speaking, optimistic version-management schemes are more suitable for handling the applications whose transactions are most likely to commit, while pessimistic version-management schemes are optimized for abort. However, if the applications do not behave as expected, these version-management schemes will incur significant overheads that offset the performance benefits of TMs. The root cause of this problem is the significant amount of time spent by existing version-management schemes to maintain various logs that in turn lead to serial accesses to the logs and incur many data movements, thus widening the isolation windows and decreasing the exposure of thread parallelism. Conflict management can schedule the conflicted transaction to avoid some pathology. However, after the comprehensive evaluation on various transactional workloads, a common conclusion is that there is no known policy that can perform universally well under all settings [32]. So it is time to exploit the potential in the version management schemes to expose more thread parallelism.

Earlier studies on TM use micro-benchmarks or parallel applications from benchmark suites like SPLASH-2 [33] to evaluate the performance. However, recent investigations into more complicated benchmark suites such as STAMP [34], Lee-TM [13], TransPlant [14], and RMS-TM [15] show that there are far more conflicts occurring in multi-threaded applications than previously assumed, resulting in more transactions being aborted due to conflicts [35]. Table I lists the abort behaviors of applications reported in published studies, which clearly suggests that, while more optimizations should be made on the expected and common commit operations, the side effects of abort operations cannot be ignored, especially for the emerging coarse-grained and high-contention applications. Lee-TM, TransPlant and RMS-TM show an increased abort ratio and that there is no absolute precedence of commit to abort, and vice versa. We believe that it is time to rethink the version-management

schemes of HTMs to adapt to the current and future coarse-grained and high-contention applications. For HTMs with optimistic version management, even the low abort ratio can degrade the performance dramatically because trapping into software to restore the old values in the undo logs consumes a lot of time before releasing the access permission. On the other hand, pessimistic version-management HTMs should optimize the commit operations because they are still the expected and common case in transactional applications.

Our analysis above indicates that existing version management schemes share a common drawback in that they require extra data movements, thus widening the isolation windows. And the high-contention application environment makes the system more vulnerable to the repair and merge pathologies (as discussed in Section 1). Therefore, we propose SUV-TM, which completely avoids extra data movements by storing the new values in the redirected locations and keeping track of the redirected mapping information in a zero-latency redirect table for commit and abort. Only one update is needed in SUV-TM whether the transaction commits or aborts, thus narrowing the isolation windows to allow more thread parallelism to be exposed and exploited. Our experimental results (see Section 5) show that this method is feasible and effective.

III. The SUV-TM Architecture

SUV decouples the version-management scheme from the undo or redo logs, by directly storing the new values of transactional write operations to the proper address and holding the mapping information in a redirect table. It can be implemented in either the eager or the lazy mode by changing the default pointer to the proper data. As we discussed earlier, commit is a more common operation than abort and thus should be optimized, which makes it suitable to implement SUV-TM in the eager mode. As SUV provides a fast switch mechanism to point to the old values on abort while saving time on maintaining the undo logs, it has a great potential to optimize the abort operation as well. Moreover, we also integrate SUV in DynTM to learn its potential in the dynamic mode(See Figure 9). Here we use the eager scheme as the case to illustrate the architecture of SUV-TM.

SUV-TM follows a two-pronged approach that uses the read/write signatures to detect conflict eagerly and the redirect table to support eager version-management, as shown in Figure 2. Read/write signatures are compact encodings of address sets accessed in transactions that are used to track the read-set and write-set. SUV-TM uses a write redirect technique to achieve eager version-management, where actual data is written to the redirected location while the redirect entry is stored in the redirect table to maintain the mapping information. Each transactional modification is required to hold both the old and new values in the original and redirected addresses until the end of the transaction. If the transaction is completed, the space holding the old values can be reclaimed for the subsequent operations without any extra space overhead or data movement except for storing the redirect entries.

The *register checkpoint module* takes snapshots of the processor so that SUV-TM has the ability to revert to the processor's state prior to the transaction. The *signature module* is used to detect transactional conflicts among different transactions. The *TM nest register* counts the transactional depth to support nesting composition. The *two-level redirect table* is used to hold redirect entries in hardware, which keeps track of the mappings between the original and the redirected addresses. A *redirect-entry pointer* is added to point to the available slot in the preserved pool, so as to easily redirect the next store operation. The overflow of the redirect table is indicated by the use of the *table overflow flag* when the two-level redirect table cannot hold all redirect entries and an extra pointer is used to point to the specific data structure for swapped out entries.

Each core in SUV-TM employs a zero-latency private redirect table to hide the latency of accessing the redirect entries while all cores in the CMP share a larger global redirect table. The redirect entries in the redirect table maintain the relationships between the original and the redirected addresses. In order to save the on-chip area cost, SUV-TM exploits the address clues on both the L1 data cache and the TLB to calculate the original and the redirected addresses. As shown in Figure 3, for example, the redirect entry represents the redirection of the original address 0x1000040 to the redirected address 0x8080. The L1 data cache set index bits are stored in the redirect entry, from which the original address can be obtained by concatenating these bits with the tag bits in cache. While SUV-TM automatically allocates a page in the preserved redirect pool to store the new values written by transactional store operations, the page mapping is stored in TLB. Thus each of SUV-TM's redirect entries stores an index to the corresponding TLB entry containing the physical page address, from which the redirected address can be obtained by concatenating it with the in-page offset. In our current prototype configuration described in Section 5, a first-level redirect entry contains 22 bits in size (i.e., 7-bit L1 cache index, 2-bit present state, 6-bit TLB index and 7-bit in-page offset). The second-level table needs more index bits than the first-level table to represent the original address. The redirect table employs a simple coherence protocol to guarantee its correctness, as detailed in the next section.

SUV-TM performs the redirect operations in a preserved memory pool whose size can be adjusted as needed. There are four states for each entry (See Table II) that help determine how past transactional redirect operations will affect future memory accesses. Two bits per entry, a *global bit* and a *valid bit*, are used to indicate these four states, where two values (with the global bit set to "1") indicate that the entry is valid to all memory accesses (inside and outside

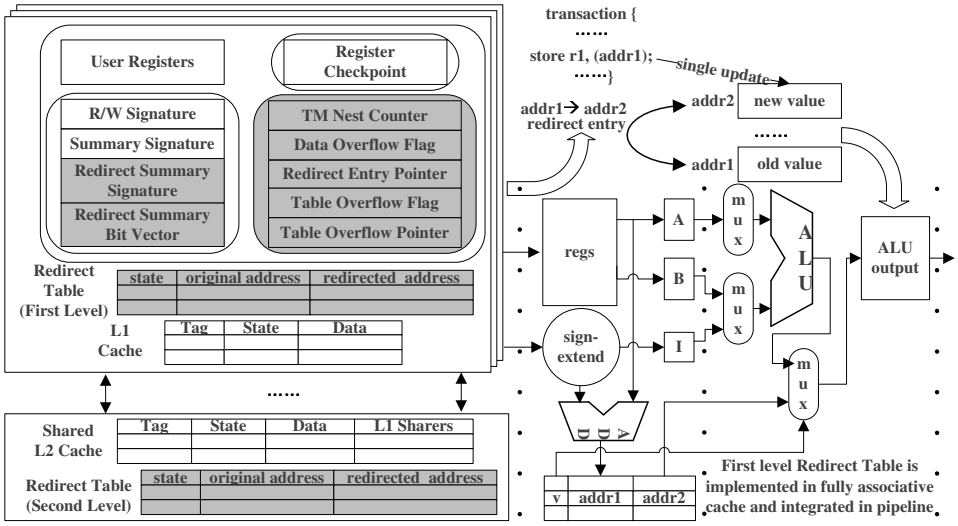


Figure 2: An Architectural View of SUV-TM

Table II: The States Represented by A Redirect Entry

State Bits		Status Explanation
G	V	
0	0	The private invalid entry
1	0	Speculatively deletes a global valid redirect entry
0	1	Speculatively adds a local valid redirect entry
1	1	The global redirect entry

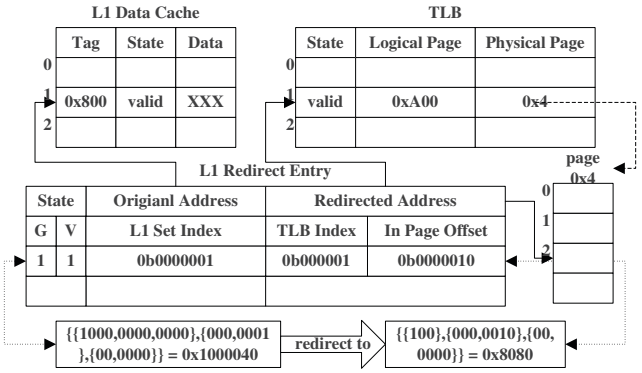


Figure 3: Addressing in A Redirect Entry

the transaction) while the other two values (with the global bit set to "0") represent the transactional transient states that only affect memory accesses within the same transaction.

The *MESI* protocol [9] is used to maintain the cache coherence, which integrates the redirect table to support the single-update version-management in SUV-TM, thus hiding the address redirection latencies from the upper memory system. A load/(store) that misses on block B generates a GETS(B)/(GETM(B)) coherence request. The core that receives a GETS(B)/(GETM(B)) request checks its write signature/(read and write signatures). When a core detects that the requested address B is in its write-set/(read-set or write-set), a conflict occurs and a NACK is sent to the requester. Besides checking the conflicts, the receiving core will check the redirect table to get the redirected address because the original address B may have been previously redirected. Upon receiving the NACK message, the requesting core resolves the conflict by stalling or aborting the transaction. An alternative policy is to make the receiving core stall or abort its transaction to guarantee the execution of the requester's transaction. If there is no conflict, the receiving core will acknowledge the requester and piggyback the redirected location in the subsequent message/(while the store operation will book the mapping information in its redirect table). After this, the requester can load/(store) data from/(to) the right address whether the address B is redirected or not. When the transaction commits, the local valid redirect entries (with the global bit set to "0"and the valid bit set to "1") will be converted to global valid entries (with the global bit set to "1"and the valid bit set to "1"). If the transaction aborts, the local valid redirect entries (with the global bit set to "0"and the valid bit set to "1") will be converted to local invalid entries (with the global bit set to "0"and the valid bit set to "0").

IV. Design and Implementation of SUV-TM

SUV-TM is an eager version-management HTM that is based on the framework of, but aims to significantly improve over LogTM-SE. As a result, SUV-TM has inherited several LogTM-SE's proven and effective features such as nested transactions, thread suspension/migration, etc. The novelty of SUV-TM lies in the way it decouples managing the old and new values of transactional modifications from the undo logs while providing a fast switch to the proper data without actual data movements. SUV-TM is transparent to the upper level OS that sees an intact existing memory hierarchy. In what follows we will illustrate the main operations of SUV-TM that realize this transparency.

A. Redirect Entry Issues

SUV-TM uses redirect entries to hold the mapping information to manage both the old and new transactional data values. The redirect entry whose structure is described in

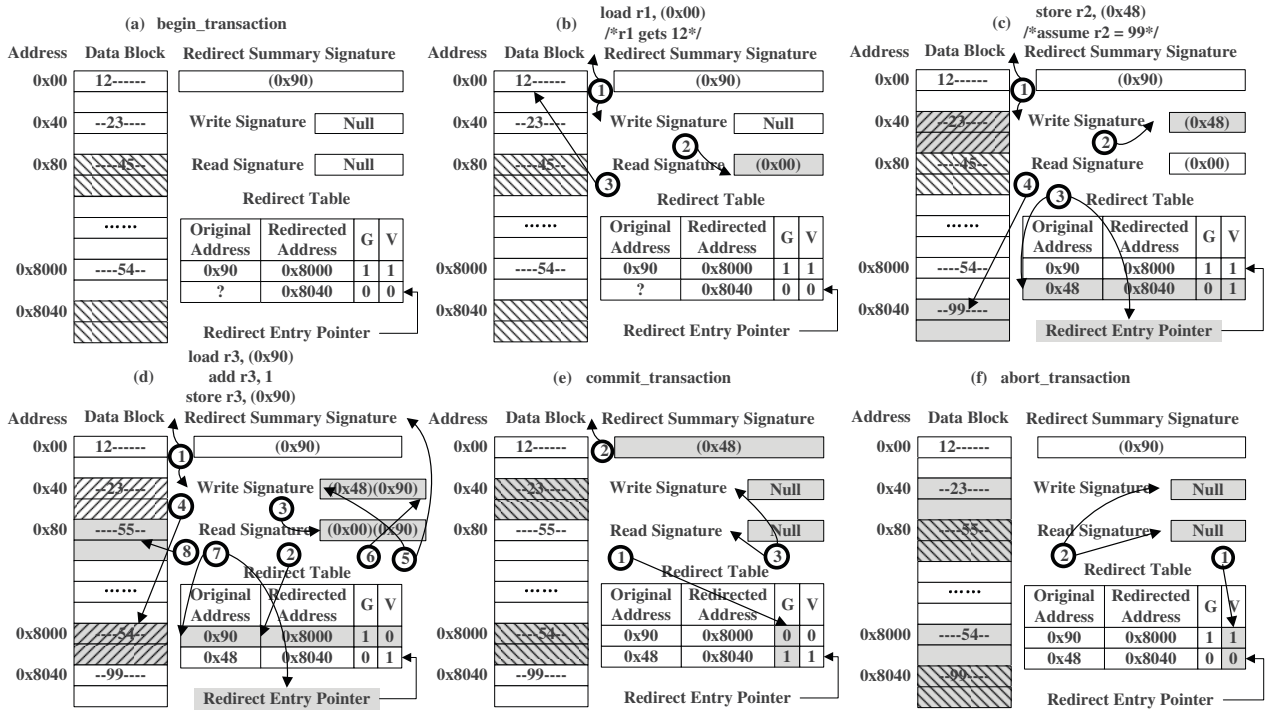


Figure 4: Basic Operations in SUV-TM

Figure 3 is a key data structure in SUV-TM that requires a coherence protocol to guarantee the correctness across the CMP. Since each entry must be in one of the four states (i.e., two stable states and two transient states), a simple write invalidate protocol like MSI is sufficient to maintain the coherence. SUV-TM incorporates a number of optimizations, such as address filtering, to speed up the address translation on each memory access in order to support strong isolation because it lies on the critical path.

When transactions commit successfully, their redirect entries are added to the redirect table to direct subsequent accesses to the right addresses. Each memory access (including non-transactional access) must look up the redirect table first to get the actual address (e.g., using the redirected address if the original address has been redirected previously). The two-level redirect table implemented in hardware is used to hold the frequently accessed entries during the execution. Our design aims to achieve a high hit rate by exploiting access locality while effectively dealing with the possible overflow of the redirect table. The first-level table is privately integrated in the existing pipeline of each core and implemented in a fully-associative search circuitry to provide the zero-latency access if the requested redirect entry hits in the first-level table. Due to the limited capacity of the first-level tables, all cores on the chip share a second-level table to hold more redirect entries at the cost of extra latency to access the second-level table. When the two-level table cannot hold all redirect entries, it can swap out some cold entries to the main memory. This routine is managed by software to guarantee the completeness of SUV-TM.

Intuitively, rapid increase in redirect entries under long-running applications may exhaust the table, thus leading to a perpetual overflow. However, if other threads update the shared variables again, the original space of the shared variables can be used to store the newly updated values if others do not occupy the original space. In this case, no redirect entry is needed because the latest redirection leads to the original space. We exploit this feature to reduce the number of redirect entries.

SUV-TM allocates a reserved memory space to store the new values of transactional writes and automatically allocates a new page on demand. SUV-TM collects the redirected data in the proper page, and reclaims the original addresses for subsequent redirect operations. As mentioned earlier, in a shared-memory environment, if the same variable is updated again, it can be redirected back to the original address, which is a very useful feature in SUV-TM.

Apart from the concern on the redirect table overflow, another problem is the extra lookup operation on the operational path of each memory access. We address this problem by using a redirect summary signature to represent the set of already redirected addresses. The redirect summary signature can filter out the un-redirected addresses quickly without any lookup operation. Nevertheless, false positives on signature may incur extra lookup, which affects the performance but not the correctness. Moreover, it is further found that false conflicts account for a large portion of the total conflicts, suggesting that it is possible and beneficial to speculatively use the original address to access memory in advance before searching the swapped out entries in main memory on a redirect-table miss, a relatively rare event that occurs under coarse-grained applications. Should the

speculation turn out to be wrong (i.e., a valid swapped out entry is found in the main memory), the transaction must be notified to discard the wrong speculative execution and re-execute the related instructions. This aggressive method can help exploit more thread parallelism.

B. Redirect Operations of SUV-TM

In addition to the aforementioned operations, transactional accesses are needed to maintain the necessary transactional information to guarantee its semantics. During each transaction, SUV-TM must record the read and write sets to detect conflicts and store both the old and new values of the data for commit or abort respectively. Meanwhile, it must add (sometimes delete) a redirect entry for each transactional write and make it valid only in this transaction before the transaction commits or aborts. Other important operations in each transaction include taking/restoring a snapshot of the processor, recording nesting information (depth and stacked frame structure), etc. In order to describe these operations clearly, we use an illustrative example in Figure 4 to delineate the operations in different stages and their corresponding transitions in a transaction. Note that some trivial flags such as TM nest, Table overflow, etc., are omitted for brevity and clarity. In each sub-figure, the left part presents memory's current state, indicating the addresses and their content, and back-slashed slots label the preserved pool for redirection. SUV-TM detects conflicts at the granularity of a cache-line (i.e., 64 bytes), so all memory accesses are aligned by 64 bytes. The right part of each sub-figure presents the read/write signature to detect the conflicts, the redirect summary signature to filter addresses quickly, and the redirect table to manage redirect entries. The gray-shaded areas in each sub-figure indicate changes on them from the previous state. This figure uses the address sets to represent the corresponding signature without the cluttering of false conflicts for clarity. The treatment of false conflicts is explained in the discussion of Figure 4.

The **begin_transaction** instruction initiates a transaction by taking a checkpoint, flushing the read and write signatures and incrementing the TM nest counter. As shown in Figure 4(a), there is a redirect entry, left by the previous transactions, redirecting the original address *@0x90* to the target address *@0x8000*, which means that the accesses to *@0x90* will be redirected to *@0x8000*.

Figure 4(b) shows the behavior of an un-redirected transactional load operation. In Step 1 the redirect summary and write signatures are first checked to determine whether this access needs to look up the redirect table or use the original address to load data directly, where in this case the latter is true and no table lookup is needed. It is followed by adding address *@0x00* to the read signature in Step 2 and loading data *12* from address *@0x00* to r1 in Step 3. If a false positive occurs during Step 1 or this address has already been redirected, SUV-TM must use *0x00* as the key to look

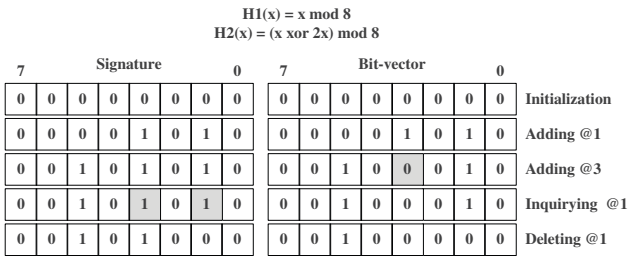


Figure 5: Operations on The Redirect Summary Signature

up the redirect table to determine exactly which case it is. In case of a false conflict, the subsequent operations will follow the process described in Steps 2 and 3 of Figure 4(b). Otherwise SUV-TM will use the redirected address to access memory as indicated in Step 2 of Figure 4(d). An un-redirected transactional store operation in Figure 4(c) takes similar actions to those described in Figure 4(b), with the exception that it adds a new redirect entry in Step 3 and updates the redirect entry pointer and stores *99* in *@0x8040*.

Figure 4(d) presents more complicated redirected transactional load and store operations. SUV-TM finds that address *@0x90* has been redirected before in Step 1 and then searches the redirect table to find the redirect entry in Step 2, which is followed by Step 3 to update the read signature and then load data *54* to r3 from the redirected address *@0x8000* in Step 4. Once r3 is changed from *54* to *55*, Steps 5 - 8 repeat the process of Steps 1 - 4 except that the former executes the store operation instead of the load operation. More specifically, for the store operation from r3(55) to *@0x90*, it finds that *@0x90* has been redirected after inquiring the redirect summary signature, then it redirects (*@0x90->@0x8000*, *@0x8000->@0x90*) back to the original address. It performs delete-entry and add-entry operations on the same entry in this case.

The **commit_transaction** instruction marks the end of the transaction and makes all transactional modifications valid and visible globally. In Figure 4(e), transient states are converted to globally valid states in Step 1 by changing the global bit from *"0"* to *"1"* if the valid bit holds a value *"1"*, or changing the global bit from *"1"* to *"0"* if the valid bit holds a value *"0"*. Then, SUV-TM updates the redirect summary signature by adding or removing the corresponding addresses in Step 2. New addresses are added by executing the *OR* operation between the redirect summary signature and the write signature. On the other hand, removing addresses from the redirect summary signature is more complicated due to the *false positive* problem resulting from the hashed signature representing a superset of the original addresses. To address this problem, we add another bit vector to record which bits are only written once from hashing operations in the redirect summary signature. An address is removed from the signature by unsetting its unique bits in the redirect summary signature that works as a Bloom Counter [36], as shown in Figure 5. It must be noted that removing addresses incompletely will not impact

Table III: Configuration of The Simulated CMP System

Processor Core	**1.2 GHz in-order, single issue**
L1 Cache	**32 KB 4-way, 64-byte line, write-back, 1-cycle latency**
L2 Cache	**8 MB 8-way, write-back, 15-cycle latency**
Main Memory	**4 GB, 4 banks, 150-cycle latency**
L2 Directory	**Bit vector of sharers, 6-cycle latency**
Interconnect	**Mesh, 2-cycle wire latency, 1-cycle route latency**
Signature	**2 Kbit Bloom filters**
1^{st} Level Table	**512-entry zero-latency fully associative table**
2^{nd} Level Table	**10-cycle latency 16384-entry 8-way shared table**

the correctness of the redirect summary signature because it is allowed to present a superset of redirected addresses, although it will induce more overheads due to the wasteful table lookups. Finally, SUV-TM flushes the read and write signatures, decrements the TM nest counter and discards the checkpoint in Step 3.

The **abort_transaction** instruction is used to abort the transaction, which entails discarding the speculative results and reverting to the previous state. As shown in Figure 4(f), SUV-TM changes all transient entries to "stable"as in the pre-transaction state, by changing the valid bit from *"0"* to *"1"* if the global bit holds a value *"1"*, or changing the valid bit from *"1"* to *"0"* if the global bit holds a value *"0"*. Finally, SUV-TM flushes the read/write signature, decrements the TM nest counter, and restores the checkpoint.

C. Other Operations of SUV-TM

To deal with the transactional nesting, we adopt the method proposed by the LogTM-Nested method [37], which uses a stacked frame to save the checkpoint, read and write signatures in the thread's private space while maintaining a nest relationship among these data. As a result, a nested transaction is executed as normal, reverting to the outer transaction's data at the end of the current transaction. This approach can easily support closed nested transactions with partial abort. It can also be extended to support open nested transactions to free more threads by registering both commit and compensate actions and handing over the compensate jobs to the parent transaction when the inner transaction commits successfully. When already committed inner transactions must be aborted because their parent transactions are aborted, it is the parent transaction's responsibility to notify the affected transactions and guarantee a consistent view.

In order to support context switching, SUV-TM copies the conflicting state to the logs. When a thread running a transaction is suspended by the scheduler, the new thread on the same core starting another transaction must check conflict with the suspended transaction. The summary signature used in LogTM-SE is sufficient to cope with this problem.

V. Performance Evaluation and Analysis

In this section, we evaluate the performance of SUV-TM by comparing it with LogTM-SE [9], FasTM [10] and DynTM [30] through extensive execution-driven experiments under various workloads. Further, we discuss the benefits of the SUV-TM design philosophy relative to FasTM and DynTM, the latest progresses in version-management reported in the literature. We also examine the sensitivity of important SUV-TM design parameters to performance. Finally, we analyze the hardware and performance overheads incurred by SUV-TM.

Table IV: Workload Characteristics of Benchmarks

	Input Parameters	Length	Contenion
bayes	**-v32 -r1024 -n2 -p20 -s0 -i2 -e2**	**43K**	**High**
genome	**-g256 -s16 -n16384**	**1.7K**	**High**
intruder	**-a10 -l4 -n2038 -s1**	**237**	**High**
kmeans	**-m40 -n40 -t0.05 -i random-n2048-d16-c16.txt**	**106**	**Low**
labyrinth	**-i random-x32-y32-z3-n64.txt**	**317K**	**High**
ssca2	**-s13 -i1.0 -u1.0 -l3 -p3**	**21**	**Low**
vacation	**-n4 -q60 -u90 -r16384 -t4096**	**2.1K**	**Low**
yada	**-a20 -i 633.2**	**6.8K**	**High**

A. Experimental Environment and Workloads

To evaluate SUV-TM, we set up a 16-core CMP system with TM functionalities based on GEMS 2.1 [38] in the Simics simulation infrastructure [39]. In particular, these cores are interconnected in a mesh topology via 64-byte links and adaptive routing. Four memory controllers are configured to access the main memory. The detailed configuration parameters are listed in Table III.

We choose LogTM-SE [9] with the same configuration as the baseline system since it is open source, publicly available, and considered the state-of-the-art scheme. In order to compare with the latest progress in version-management schemes, we implement FasTM and DynTM to assess SUV-TM against the latest and state-of-the-art version-management schemes. For simplicity and objectivity, we adopt the ***Stall policy*** (Stalling the requester and avoiding any possible cyclical dependence among those stalled transactions) to resolve conflicts among transactions. Many existing studies on LogTM also adopt this policy, which enables indirect comparisons with other schemes [9, 10, 40].

STAMP [34] is the first comprehensive benchmark suite designed specifically for TM in that programmers' habits in parallel programming are considered and integrated. It is also easy to vary different parameters in STAMP to conduct comprehensive evaluations. The key workload characteristics we use throughout this paper are summarized in Table IV.

B. Experimental Results

Existing TMs including LogTM-SE have been proven to have better, or at least comparable, performance than lock mechanisms [7, 9, 41], so we focus on the performance improvement gained by SUV-TM over LogTM-SE, FasTM and DynTM.

To obtain a comprehensive understanding of the performances of various TMs, we break the execution time

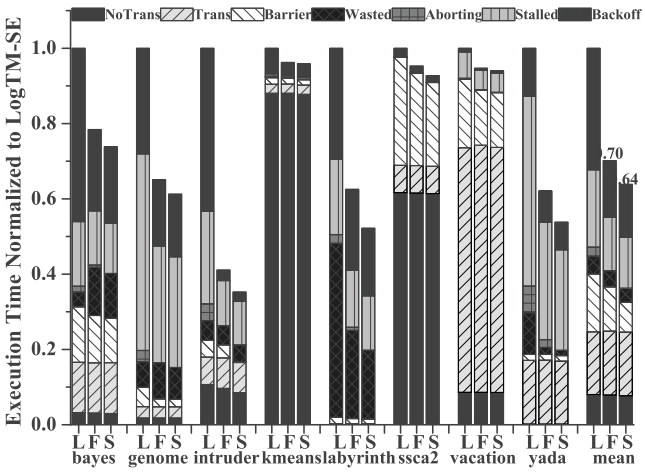


Figure 6: Distribution of The Execution Times of LogTM-SE (L), FasTM (F) and SUV-TM (S)

down to these components: time due to non-transactional work (*NoTrans*), time due to un-stalled transactional work (*Trans*), time waiting on a barrier (*Barrier*), time stalling after an abort (*Backoff*), time stalling to resolve conflict (*Stalled*), time due to wasted work when a transaction is aborted (*Wasted*), and time due to rolling back during abort (*Aborting*). The first three components, *NoTrans*, *Trans*, and *Barrier*, are necessary costs while the remaining, *Backoff*, *Stalled*, *Wasted*, and *Aborting*, are extra overheads of serializing transactions.

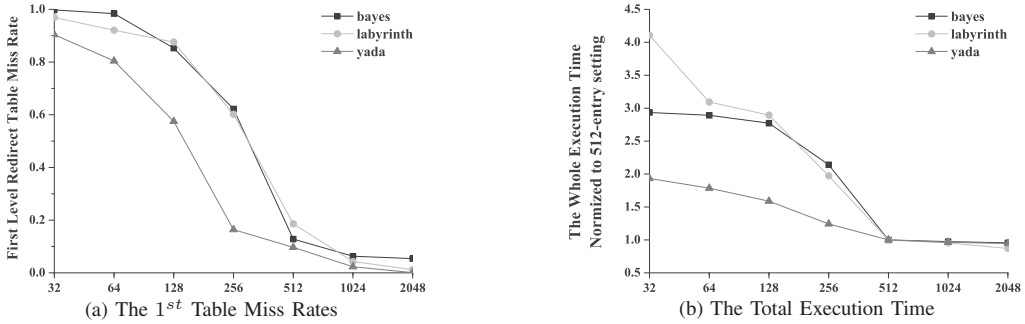
Before presenting the experimental results of LogTM-SE, FasTM and SUV-TM, let us briefly review their version-management mechanisms. LogTM-SE uses the undo logs to hold old values while updating in-place to implement eager version-management, and it traps into software to restore the old values on abort. FasTM modifies the cache coherence protocol on top of LogTM-SE to improve the undo log technique on managing transactional modifications by exploiting the data inconsistency among different levels in the memory hierarchy. It first writes back the dirty data in the L1 cache to the lower-level memory, keeping the new values in the L1 cache and the old values in the lower-level memory, but degenerates to LogTM-SE when the L1 cache overflows. SUV-TM directly stores the new values in redirected addresses, keeps the mapping information of the original and redirected addresses in the redirect table, switches to the proper version at the end of each transaction.

Figure 6 shows a significant performance improvement by SUV-TM and FasTM over LogTM-SE, where SUV-TM outperforms FasTM. FasTM and SUV-TM's advantages are more pronounced under high-contention applications such as $bayes$, $genome$, $intruder$, $labyrinth$ and $yada$. The main reasons behind SUV-TM's superiority to LogTM-SE are fourfold. First, SUV-TM removes the *Aborting* time for the most part except in the relatively rare event when the redirect entries miss in the first-level table. Second, the redirected write operation shortens the time spent on version-management (without data movement), thus narrowing the isolation windows and reducing the time spent on *Trans* and *Wasted*. Third, releasing the isolation window early lessens conflicts by allowing more transactions to execute simultaneously, resulting in less time being spent on *Stalled*, *Wasted* and *Backoff*. Finally, the redirect table speeds up memory access by reducing the transactional overflows, which in turn lowers the overheads. FasTM achieves a good performance gain over LogTM-SE because it removes most of the *Aborting* time if the L1 cache does not overflow. Based on a similar idea, SUV-TM shortens the duration of holding the exclusive access permission to expose more thread parallelism by freeing more blocked threads. Moreover, besides removing the *Aborting* time, SUV-TM uses a more aggressive method to reduce the time spent on maintaining the undo logs, further optimizing both commit and abort operations and thus exploiting more parallelisms than FasTM. An important feature distinguishing SUV-TM from FasTM is that the former does not degenerate to LogTM-SE when the L1 cache overflows as the latter does, meaning that, while FasTM may fail to cope with the future coarse-grained and high-contention applications, SUV-TM is more suitable for such workloads. Another reason why SUV-TM outperforms FasTM is that the former does not need to write back the shared dirty data in the L1 cache before starting a new transaction and avoids extra accesses to the low-level memory to fetch the old value on abort.

Table V lists statistic information about the overflows, which indicates that the redirect table avoids nearly half of the transactional data overflow in these applications. LogTM-SE and FasTM suffer from transactional overflow severely while SUV-TM mitigates the transactional overflows but incurs some redirect table overflows under these three coarse-grained applications. This is because the redirect operations can be used to store data from frequently accessed addresses to other addresses in order to reduce conflicts on the hottest addresses. Although the write-set of each overflowed transaction is big, there is a strong access locality among the transactions. We find that the working set of these three applications are less than 8MB. In addition to transactional data cache overflows, some large transactions in the $bayes$, $labyrinth$ and $yada$ benchmarks with huge write-sets also overflow the redirect table, thereby resulting in extra latency on accessing to the redirect entries residing in the second-level table or the main memory. Fortunately, this is shown to be a rare case throughout our extensive evaluations. This feature, repeatedly updating the value of the shared variable among different transactions (i.e., writing to a shared variable in transaction $t1$ redirects the original address $@A$ to the redirect address $@R$, later writing to the same variable in transaction $t2$ redirects the actual address $@R$ back to the original address $@A$), can effectively reduce the number of redirect entries. So the redirect table overflow rate is rather low even under coarse-

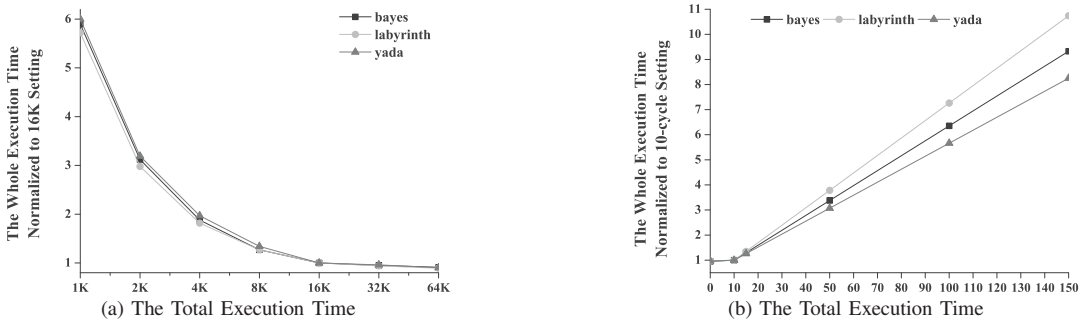
Table V: Statistics on The Overflows for $bayes$, $labyrinth$ and $yada$

	Overflow footprint		Cache Overflow Rate			Redirect Table Hit Rate			The Total Overheads		
	Read-Set	Write-Set	LogTM-SE	FasTM	SUV-TM	L1 Table	L2 Table	Memory	LogTM-SE	FasTM	SUV-TM
bayes	**223**	**190**	**22.2%**	**21.6%**	**13.1%**	**87.2%**	**10.1%**	**2.7%**	**17.1%**	**15.8%**	**8.8%**
labyrinth	**222**	**225**	**40.0%**	**38.5%**	**19.2%**	**81.4%**	**15.2%**	**3.4%**	**11.2%**	**10.4%**	**3.5%**
yada	**256**	**184**	**15.7%**	**15.3%**	**8.7%**	**90.3%**	**9.5%**	**0.2%**	**7.8%**	**6.1%**	**3.2%**



(a) The 1^{st} Table Miss Rates (b) The Total Execution Time

Figure 7: Sensitivity to The Size of The 1^{st} Redirect Table



(a) The Total Execution Time (b) The Total Execution Time

Figure 8: Sensitivity to The Size and Latency of The 2^{nd} Redirect Table

grained applications. FasTM degenerates to LogTM-SE on the L1 cache overflow, with the overall performance being adversely affected by abort operations. SUV-TM's advantage over FasTM is likely to be more pronounced with coarse-grained and high-contention transactions.

To better understand the first-level redirect table overflow and its impact, we conducted a sensitivity study to quantify the impact of the first-level redirect table miss rate and the total execution time by varying the size of the first-level redirect table. Then we further vary the size and the access latency of the second-level redirect table to learn their impacts on the total execution time.

Figure 7(a) clearly indicates that a 512-entry first-level redirect table can achieve a high hit rate with a relatively low space and time cost. There is almost no improvement on the execution time by scaling the table size beyond 512, as shown in Figure 7(b). Our extensive experimental results show that SUV-TM is effective and efficient in handling most transactions with a moderate implementation cost.

Figure 8(a) shows that the execution time decreases with the increasing size of the second-level table, but the performance gain is insignificant when the size rises beyond 16K. Figure 8(b) shows that the execution time will dramatically increase when the latency of the second-level table rises beyond 10 cycles and the zero-latency of the second-level table improves the performance by less than 5%.

Finally, we integrate the SUV scheme into DynTM to replace its original version management scheme adopted from FasTM to compare their performance in a flexible framework. DynTM [30] uses a history-based selector to choose either the eager version management and eager conflict management scheme or the lazy version management and lazy conflict management scheme to execute each transaction. So it will iccur some time overhead spent on committing, which is labeled as "Committing" in Figure 9 while the other parts of the latency are defined the same as that in Figure 6. As shown in Figure 9, DynTM with the SUV scheme as its version management scheme outperforms the original DynTM with the FasTM version-management scheme, with an average speedup of 9.8% across the 8 applications of the STAMP benchmark suite. More importantly, the performance advantage of DynTM with SUV over the original DynTM becomes more significant under the 5 high-contention and coarse-grained applications of the STAMP benchmark suite, achieving an average speedup of 18.6%, because the former can flexibly point to either the old or new copy data with the help of the prediction results from the history-based selector.

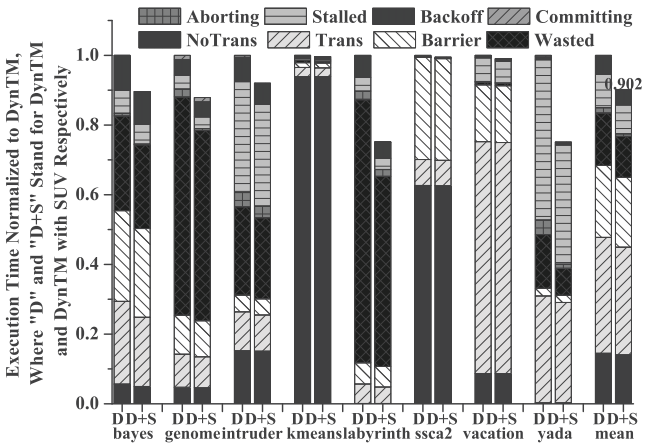


Figure 9: Distribution of The Execution Times of The Original DynTM (D) and The DynTM with SUV as Its Version-Management Scheme (D+S)

Table VI: Parameters of Some Contemporary Processors

Processor	Tech (nm)	Clock (GHz)	Cores/ Threads	TDP (W)	Area (mm^2)
UltraSPARC T1 [42]	90	1.4	8/32	72	378
UltraSPARC T2 [42]	65	1.4	8/64	84	342
Rock Processor [2]	65	2.3	16/32	250	396

C. Complexity of SUV

SUV adds a zero-latency, 512-entry fully-associative first-level redirect table, a $2K$-bit redirect summary signature and a $2K$-bit vector for each core, and a 10-cycle latency 16K-entry 8-way second-level redirect table shared across the CMP. Each core needs 1.9KB (i.e., (2Kb + 2 Kb + 22b x 512/1024)/8 = 1.875KB) memory element, which is about 5.86% of the L1 data cache (32KB) capacity of modern processors. The area cost of the shared second-level redirect table is not a big problem considering the size of the L2 cache (i.e., usually several megabytes).

To better understand the extra overheads of the 512-entry fully-associative redirect table on access time, energy consumption and silicon area, we estimate these overheads by CACTI 5.3 [43] and find that it is feasible in hardware implementation. In order to compare with the state-of-the-art processors, we gather some contemporary processors' parameters in Table VI, and list the results of the 512-entry fully-associative table from CACTI in Table VII. Due to the CACTI requirement that the minimum size of a line be 8-byte, we set a 4KB 512-entry fully-associative table to obtain its overheads and we believe that the actual SUV overheads should be less than half of the estimates given by CACTI because each entry in the SUV redirect table occupies only 22-bit instead of the 64-bit required by CACTI. From Table VII, an access to the fully-associative table can be finished in 1 cycle with the 45 nm CMOS process at 1.2GHz, meanwhile, the read/write energy consumption is 0.150/0.163 nj, so the maximum extra energy consumption of the fully-associative table in the simulated CMP is less than 0.5 x (0.150 nj + 0.163 nj) x 16 x 1.2 x 10^9 = 3 j, which is about 1.2% of the Rock processor power consumption. Moreover, the silicon area required by the fully-associative table (i.e., 0.5 x 16 x 0.282 = 2.26 mm^2), which is 0.6% of the Rock processor silicon area, is also acceptable.

Table VII: Overheads of The First-Level Fully-Associative Table Estimated by CACTI

Tech (nm)	Access Time (ns)	Dynamic Energy (nj)		Area (mm^2)
		Read	Write	
90	1.382	0.403	0.434	0.951
65	0.995	0.239	0.260	0.589
45	0.588	0.150	0.163	0.282
32	0.412	0.072	0.078	0.143

In order to support the strong isolation to detect the conflicts between transactional accesses and non-transactional accesses, the redirect table lookup operation lies on the critical path of memory access operations. It will necessarily slow the non-transactional accesses, if a miss occurs in the first-level table. Fortunately, the miss rate of the first-level table is rather low even under coarse-grained applications. This side effect is relatively insignificant and we consider it a moderate overhead required to support strong isolation.

VI. Conclusion

To address the observed repair and merge pathologies, we propose SUV, a novel single-update version-management scheme, that can take advantage of a redirect table to provide zero-latency commit and abort operations. Redirecting each transactional store to a redirected address eagerly requires only one data movement regardless of commit or abort. Removing execution time on commit and abort shortens isolation windows, thus reducing the forced serialization among multiple transactions and enabling more transactional parallelism to be exploited.

We have evaluated SUV in SUV-TM by comparing it with the state-of-the-art LogTM-SE, FasTM and DynTM schemes, through extensive execution-driven experiments under the STAMP benchmark suite that represents a wide spectrum of high-contention applications. SUV-TM shows significant performance advantages, outperforming the state-of-the-art LogTM-SE and FasTM schemes by 56% and 9% respectively for all applications of the STAMP benchmark suite, and 95% and 12% respectively for the five high-contention applications of the STAMP benchmark suite. Further, the experimental results show that the DynTM integrated with SUV as its version-management scheme outperforms the original DynTM based on FasTM by 9.8% and 18.6% respectively for all applications and the five high-contention applications of the STAMP benchmark suite. Moreover, we use CACTI to evaluate the hardware overheads of SUV and find that it is feasible in hardware implementation.

Acknowledgements

We would like to thank the anonymous reviewers for their suggestions and comments. This work was supported by the

National Basic Research 973 Program of China under Grant No.2011CB302301, the National High Technology Research and Development Program ("863" Program) of China under Grant No.2009AA01A402, National Natural Science Foundation of China (NSFC) under Grant No.61025008, No.60933002, No.60873028 and No.61173043, the US NSF under Grants IIS-0916859, CCF-0937993, CNS-1016609, and CNS-1116606.

REFERENCES

[1] M. Herlihy and J. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993, pp. 289–300.

[2] S. Chaudhry, "Rock: A third Generation 65nm, 16-Core, 32 Thread + 32 Scout-Threads CMT SPARC Processor," in *Proceedings of the 20th IEEE International Symposium on High Performance Chips (HotChips)*, 2008.

[3] C. Click, "Azul's Experiences with Hardware Transactional Memory," in *Transactional Memory Workshop*, 2009.

[4] R. Haring, "The IBM Blue Gene/Q Compute Chip+SIMD Floating-point Unit," in *Proceedings of the 23th IEEE International Symposium on High Performance Chips (HotChips)*, 2011.

[5] "Sequoia, http://en.wikipedia.org/wiki/ibm_sequoia."

[6] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory (2nd edition)*. Morgan & Claypool, 2010.

[7] K. Moore, J. Bobba, and et.al, "LogTM: Log-based Transactional Memory," in *Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, 2006, pp. 254–265.

[8] W. Chuang, S. Narayanasamy, and et.al, "Unbounded Page-based Transactional Memory," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 347–358.

[9] L. Yen, J. Bobba, and et.al, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2007, pp. 261–272.

[10] M. Lupon, G. Magklis, and A. González, "FasTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009, pp. 293–302.

[11] R. Titos, M. E. Acacio, and J. M. Garcia, "Speculation-based Conflict Resolution in Hardware Transactional Memory," in *Proceedings of the 23rd IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2009, pp. 1–12.

[12] S. Jafri, M. Thottethodi, and T. Vijaykumar, "LiteTM: Reducing Transactional State Overhead," in *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2010, pp. 1–12.

[13] M. Ansari, C. Kotselidis, and et.al, "Lee-TM: A Non-trivial Benchmark for Transactional Memory," in *Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2008, pp. 196–207.

[14] J. Poe, C. Hughes, and T. Li, "TransPlant: A Parameterized Methodology for Generating Transactional Memory Workloads," in *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009, pp. 1–10.

[15] G. Kestor, S. Stipic, and et.al, "RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications," in *TRANSACT '09: The 4th Workshop on Transactional Computing*, 2009.

[16] Z. Yan, D. Feng, and Y. Tan, "Poster: Dissection The Version Management Schemes in Hardware Transactional Memory Systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 39, pp. 78–78, September 2011.

[17] L. Hammond, V. Wong, and el.al, "Transactional Memory Coherence and Consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004, pp. 102–113.

[18] M. Lupon, G. Magklis, and A. González, "Version Management Alternatives for Hardware Transactional Memory," in *Proceedings of the 9th Workshop on MEmory performance: DEaling with Applications, systems and architecture (MEDEA)*, 2008, pp. 69–76.

[19] C. Blundell, J. Devietti, and et.al, "Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 24–34.

[20] J. Bobba, N. Goyal, and et.al, "TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008, pp. 127–138.

[21] A. McDonald, "Architectures for Transactional Memory," Ph.D. dissertation, Stanford University, June 2009.

[22] C. Ananian, K. Asanovic, and et.al, "Unbounded Transactional Memory," in *Proceedings of the 11th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2005, pp. 316–327.

[23] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing Transactional Memory," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005, pp. 494–505.

[24] A. Shriraman, M. Spear, and et.al, "An Integrated Hardware-Software Approach To Flexible Transactional Memory," in *Proceedings of the 34th Annual International Symposium on Computer architecture (ISCA)*, 2007, pp. 104–115.

[25] S. H. Pugsley, M. Awasthi, and et.al, "Scalable and Reliable Communication for Hardware Transactional Memory," in *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 144–154.

[26] A. Shriraman, S. Dwarkadas, and M. Scott, "Flexible Decoupled Transactional Memory Support," in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA)*, 2008, pp. 139–150.

[27] D. Dice, Y. Lev, and et.al, "Early Experience with a Commercial Hardware Transactional Memory Implementation," in *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009, pp. 157–168.

[28] J. Chung, C. Minh, and et.al, "Tradeoffs in Transactional Memory Virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 371–381.

[29] B. Khan, M. Horsnell, and et.al, "An Object-Aware Hardware Transactional Memory System," in *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2008, pp. 93–102.

[30] M. Lupon, G. Magklis, and A. González, "A Dynamically Adaptable Hardware Transactional Memory," in *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010, pp. 27–38.

[31] A. Armejach, A. Seyedi, and et.al, "Using a Reconfigurable L1 Data Cache for Efficient Version Management in Hardware Transactional Memory," in *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2011, pp. 361 –371.

[32] W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2005, pp. 240–248.

[33] M. Woo, S.and Ohara and et.al, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, 1995, pp. 24–36.

[34] C. Minh, J. Chung, and et.al, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Proceedings of the 4th IEEE International Symposium on Workload Characterization (IISWC)*, 2008, pp. 35–46.

[35] C. Hughes, J. Poe, and et.al, "On the (Dis)similarity of Transactional Memory Workloads," in *Proceedings of the 5th IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 108–117.

[36] L. Fan, P. Cao, and et.al, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," *IEEE/ACM Trans. Netw.*, vol. 8, pp. 281–293, June 2000.

[37] M. Moravan, J. Bobba, and et.al, "Supporting Nested Transactional Memory in LogTM," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 359–370.

[38] M. Martin, D. Sorin, and et.al, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Computer Architecture News*, vol. 33, pp. 92–99, November 2005.

[39] P. Magnusson, M. Christensson, and et.al, "Simics: A Full System Simulation Platform," *IEEE Computer*, vol. 35, pp. 50–58, 2002.

[40] J. Bobba, K. Moore, and et.al, "Performance Pathologies in Hardware Transactional Memory," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 81–91.

[41] C. Rossbach, O. Hofmann, and E. Witchel, "Is Transactional Programming Really Easier," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2010, pp. 47–56.

[42] "Opensparc, http://www.opensparc.net/."

[43] "Cacti, http://www.hpl.hp.com/research/cacti/."