Received January 12, 2022 | Revised June 3, 2022 | Accepted July 13, 2022

DOI: 10.1002/spe.3134

RESEARCH ARTICLE

Design and Implementation of a Fully-Transparent Partial Abort Support for Software Transactional Memory

Alessandro Pellegrini[1] | Pierangelo Di Sanzo[2] | Andrea Piccione[3] | Francesco Quaglia[1]

[1]DICII, University of Rome "Tor Vergata", Rome, Italy.
[2]DISIM, University of L'Aquila, L'Aquila, Italy.
[3]DIAG, Sapienza, University of Rome, Rome, Italy.

Correspondence
Alessandro Pellegrini, DICII, University of Rome Tor Vergata, Rome, Italy.
Email: a.pellegrini@ing.uniroma2.it

Summary

Software Transactional Memory (STM) provides synchronization support to ensure atomicity and isolation when threads access shared data in concurrent applications. With STM, shared data accesses are encapsulated within transactions, which are automatically handled by the STM layer. Hence, programmers are not requested to use code-synchronization mechanisms explicitly, like locking. In this article, we present our experience in designing and implementing a partial abort scheme for STM. The objective of our work is threefold: 1) enabling STM to undo only part of the transaction execution in the case of conflict, 2) designing a scheme that is fully transparent to programmers, thus also allowing to run existing STM applications without modifications, and 3) providing a scheme that can be easily integrated within existing STM runtime environments without altering their internal structure. The scheme that we designed is based on automated software instrumentation, which injects into the application capabilities to undo the required portions of transaction executions. Further, it can correctly undo also non-transactional operations executed on the stack and the heap during a transaction. This capability provides programmers with the advantage of writing transactional code without concerns about the side effects of aborted transactions on both shared and thread-private data. We integrated and evaluated our partial abort scheme within the TinySTM open-source library. We analyze the experimental results we achieved with common STM benchmark applications, focusing on the advantages and disadvantages of the proposed solutions for implementing our scheme's different components. Hence, we highlight the appropriate choices and possible solutions to improve partial abort schemes further.

KEYWORDS:
Transactional Memory, Partial Abort, Software Transactional Memory, Static Binary Instrumentation.

1 | INTRODUCTION

In the last decades, parallel architectures left the niche of scientific and high-performance computing. Today, desktops, laptops, and smartphones are equipped with multiple processors and/or multi-core processors[1]. Also, larger-scale servers with up to 64/128 CPU cores can be purchased for a few thousand dollars. To effectively exploit this increased computing power with reduced effort, programmers require efficient and easy-to-use software development tools to hide away the architectural complexity of modern parallel computing architectures.

In this context, Transactional Memory (TM)[2] offers a practical programming paradigm for the development of concurrent applications. TM relieves the programmer from the burden of writing explicit thread-synchronization code to protect shared data accesses—the portions of code to be synchronized can be simply annotated as *transactions*. Data conflicts between transactions are detected and resolved at runtime by the TM runtime environment, thus transparently guaranteeing isolation and atomicity in executing the transactional code.

TM can be implemented in software (Software Transactional Memory—STM) or hardware (Hardware Transactional Memory—HTM). With STM, transactions management is completely delegated to a software library. Differently, with HTM, it is supported via dedicated CPU instructions. After more than fifteen years of research, several STM implementations have been released[3,4,5]. Hence, STM has reached a certain degree of maturity, so that it has been also included in mainstream compilers, such as GCC[6]. Meanwhile, HTM has gained a place in commercial processors, implemented through specific instruction set extensions (like the Intel TSX[7]). It is worth noting that STM and HTM implementations are different in some relevant features, showing various advantages and drawbacks. For example, in HTM, transactions' data footprint is limited by the processor's cache size. Accordingly, transactions that need to access a more extensive data set require software support to execute successfully. Also, with HTM, many system events (such as interrupts, context switches, system calls) can lead running transactions to be aborted. All these drawbacks are absent in STM. Thus HTM cannot wholly replace STM.

On the other hand, STM requires additional research effort to address some problems that hindered their initially expected success. In particular, some relevant disadvantages of STM implementations include: 1) the performance penalty introduced by the software support to detect and resolve transaction conflicts, and 2) the lack of a mechanism to undo modifications also made to thread-private data when a transaction is aborted and restarted. In particular, the latter aspect requires the programmer's care to write transactional code that is idempotent in writing thread-private data, thus increasing the complexity level. Concerning the performance issue, a great effort has been made by researchers to limit the performance impact of STM and to improve the STM performance through various approaches, spanning from alternative conflict detection and resolution algorithms[8,9,10,4,11], to dynamic concurrency control and dynamic tuning[12,13], and to transaction scheduling techniques[14,15,16,17,18,19,20,21,22].

However, one approach that has been less explored is *partial abort* of transactions, particularly with the goal of designing effective runtime mechanisms to make this approach viable and transparent on off-the-shelf systems. Partial abort avoids that, in the case of conflict, the whole transaction is rolled back, allowing to restart it from some (still consistent) intermediate execution point. This avoids undoing the entire work performed by an aborted transaction and thus can restrain the final transaction execution time. Without partial abort, this latency can broadly grow primarily because of a non-minimal conflict rate. The partial abort approach can also increase fairness in the case of longer-lasting transactions that access a non-minimal amount of data and consequently show increased abort probability compared to shorter ones.

Unfortunately, typical STM implementations do not offer partial abort support, as it is hard to implement when dealing with the complexity of real-world architectures. Indeed, to allow transactions to be partially rolled back, there are a number of factors to cope with, such as identifying the correct restarting point, managing the associated execution contexts, keeping track of modified data values in different memory areas, and correctly restoring some past execution context. The latter requires restoring the CPU state and all the written transactional and non-transactional variables (both on the stack and in the heap). These are some of the factors which make it complex to achieve complete transparency towards the application programmer.

In this article, we present our experience in designing and implementing a fully transparent partial abort support tailored for off-the-shelf computer architectures. Unlike previously proposed solutions, the one we designed neither requires the programmer to identify the restoration points in the transactional code explicitly, nor limits the restoration points to some semantic-specific transaction execution points, such as logical operations on shared data structures[23] or nested transactions[24]. Further, being able to undo also write operations executed within transactions on memory areas of the stack and the heap allows the programmer to write transaction code without concerns about unexpected effects of both partially and fully aborted transactions, also on thread-private data[1]. Our study focused on implementing a scheme to fit existing STM implementations running on mainstream system architectures. Notably, our partial abort scheme does not require altering the STM runtime environment. Also, it does not require any specific architectural support for tracking modifications to data that have to be undone in the case of an abort.

The implementation of the scheme that we designed introduces minimal housekeeping overhead, relying on optimized approaches to manage partial-log operations (e.g. saving snapshots of automatic variables) during the execution of the transaction. Moreover, static and automated binary software instrumentation achieves complete transparency towards the programmer.

[1]We consider as thread-private data all local variables stored onto the stack. We do not explicitly cope with Thread Local Storage, as this is a feature that the largest part of transactional applications do not rely upon.

The remainder of this article is structured as follows. In Section 2, we discuss related work. Details about the design of our partial abort scheme are provided in Section 3. A concrete implementation of the scheme for a commonly used STM environment, i.e. TinySTM[3], is presented in Section 4. Finally, in Section 5, we present the results of an experimental evaluation study.

2 | RELATED WORK

Different approaches have been proposed in the literature to reduce the wasted processing time due to transaction aborts in TM systems. Many proposals focus on the design, optimization, and/or tuning of the conflict detection and management algorithms used by STM implementations[17,9,12,8,10,4,11]. Other studies proposed compiler-based and runtime techniques which use statistics collected at runtime to identify performance bottlenecks and to guide tuning decisions (e.g. Wu et al.[25]). Further, autonomic-computing approaches for mapping threads to the different cores in multi-core machines, as well as machine learning-based techniques for improving the performance of STM application at runtime have been studied (e.g., Xiao et al.[26], Zhou et al.[27]). However, none of these proposals specifically copes with mechanisms for aborting and rolling back transactions, but they simply assume that aborted transactions are entirely rolled back. Thread/transaction scheduling[21] represents an alternative approach reducing the transaction execution time. Thread scheduling targets the (dynamic) tuning of the number of concurrent threads of the application[15] to maximize the throughput. Transaction scheduling serializes the execution of some transactions in advance, based on some scheduling policy (e.g. transactions are serialized when the transaction conflict rate exceeds a specific threshold[14]). Essentially, they are proactive approaches that aim to prevent transaction aborts, thus complementary to partial abort techniques.

The first proposal of partial abort in the context of STM can be found in Koskinen et al.[28]. The authors propose to replace nested transactions with the notion of checkpoints and continuations. The proposed approach defines the checkpoints based on the semantic of objects accessed along the transactional code execution. Accordingly, the restoration points can only be associated with logical operations executed on objects rather than any potential conflicting transaction operation. A similar approach is also found in[29,24,30].

Other solutions targeting partial abort in STM include Luper et al.[31] and Gupta et al.[32,33]. Unlike our scheme, the proposal in Luper et al.[31] is limited to the rollback of data in the write-set of transactions; thus, it does not support the restoration of all data modified along the whole portion of the transaction to be rolled back (e.g. thread-private data). Consequently, mutual consistency between shared and private data within the partial abort scheme is demanded from the programmer. The proposals in Gupta et al.[32,33] present partial abort schemes validated only via simulation. Additionally, the techniques rely on the traditional approach where shared and private objects are marked as updated via dirty-bitmaps and are logged within per-object undo stacks. Differently, our proposal does not rely on any explicit management of dirty bitmaps. It packs log information by clustering thread-private data via optimized thread-stack log operations, based on ranges of memory addresses defining target regions for the memory write instructions.

In Le et al.[34], the authors propose a partial abort scheme in the context of functional programming compilers, leveraging the continuation-passing style (CPS) programming paradigm. Consequently, the scheme is not required to address the problem of tracking and restoring modified heap and stack objects.

In the context of database transactions, one work related to our proposal can be found in Azer et al.[35], where the notion of transaction save-point is exploited. Specifically, upon detecting a conflict, a copy of the current transaction is forked. It remains idle, thus acting as the save-point to reduce the cost of transaction aborts. Unlike this approach, our proposal does not rely on forking mechanisms, which would induce excessive overhead in STM settings (given the reduced resource requirements of STM transactions compared to database transactions due to, e.g., the absence of interactions with stable storage devices). Instead, we exploit low-cost log/restore operations at the level of the stack of the transactional thread, in combination with traditional shared-data consistency mechanisms.

As already mentioned in Section 1, the STM support is today offered by some mainstream compilers. As for GCC, it ships with `libitm` (since version 4.7), a runtime library supporting STM by relying on a global lock or read/write locks. The latter implementation is close to the Lazy Snapshot Algorithm[36]. Also, GCC allows using external STM runtime libraries, such as TinySTM or RSTM[37]. In both cases, the standard STM implementations offered by GCC do not provide partial abort facilities. On the other hand, the scheme we provide in this article can also be used with GCC by relying on compile-time flags.

Concerning the management of mutual access to data structures stored in the heap, several proposals in the literature address this issue in the context of speculative high-performance computing. The works in Toccaceli et al.[38] and Pellegrini et. al.[39]

provide transparent logging facilities for dynamic memory. However, every time a snapshot is created, this entails copying the whole used heap, which is a cost too high to be paid in the context of STM systems, especially when transactions are very short. Some proposals, such as Quaglia et al.[40] and Fujimoto et al.[41], rely on specialized hardware, while we target off-the-shelf machines. Steinman[42] and Das and Fujimoto[43] propose solutions to checkpoint dynamic memory, although they are not transparent since the application must be developed by relying on an ad-hoc allocation/deallocation API. The work in Pellegrini et al.[44] offers a transparent and incremental checkpointing mechanism of dynamic memory, relying on software instrumentation. As we will show, we adapt these solutions to the TM context, in which the duration of a transaction can be very short. Moreover, solutions like the ones provided in[42,43,44] do not allow any partial rollback capability, because they undo the execution of a whole atomic unit of work (i.e., the execution of an event handler).

3 | THE PARTIAL ABORT SCHEME

Our partial abort scheme targets the widely-known Commit-Time-Locking (CTL) concurrency control algorithm, used to manage concurrency and determine consistency in typical STM implementations such as TinySTM[3] and TL2[45]. Our approach exploits the CTL read validation mechanisms and supports transparently injected in the STM-based application via static software instrumentation. Additionally, it introduces transparent mechanisms to save parts of the transactional work also related to variables in the heap and private variables in the stack. In the rest of this Section, we provide an overview of CTL, and we then plunge into the details of our partial abort scheme.

3.1 | Target Concurrency Control Algorithm

The target concurrency control algorithm is the Commit-Time Locking with read validation. We focus on the implementation provided in TinySTM[3]. It relies on a *global version clock* (*gvc*), which is a globally-shared counter which is atomically incremented when a transaction that updated some shared object commits. Each shared object (which can correspond to one or more memory words in the case of word-based STMs, like in TinySTM) is associated with its own metadata, containing a lock-bit and a timestamp. The association is established through a hash function which takes in input the memory address of the shared object and returns the memory address of the associated metadata. When a transaction successfully commits, the new value *gvc* is stored in the timestamps of metadata associated with all the objects written by the transaction.

When a thread starts a transaction or restarts a transaction from the beginning after an abort, the current value of the *gvc* is stored in a thread-local variable, called the transaction's start timestamp (*tst*).

When executing a transactional read operation on a shared object, the object's memory address is added to the transaction's read-set. In addition, it is checked in advance if the current transaction has already written the object by looking into the transaction's write-set. If this is the case, the value stored in the write-set is returned. If the shared object is not in the write-set, the object's lock-bit and a timestamp are atomically read. Then it is checked whether the lock-bit is set, meaning that a concurrent transaction locks the object. In the positive case, the transaction starts spinning until the lock-bit is found not set. Then, the object value and its timestamp are re-sampled along with the lock-bit to check if the timestamp is less than or equal to the *tst* of the reading transaction and if the lock-bit has not been set by another transaction meanwhile. If both checks succeed, thus no concurrent transaction has modified the object between the start of the transaction and the read operation, the read value is still *valid* and the transaction continues its execution. This check is called *read validation*. If the read validation fails, the transaction is aborted and restarted. When executing a transactional write operation, the address of the object to update is added to the transaction's write-set, and the value is not immediately written on the object. Instead, it is temporarily stored in a buffer in the write-set.

Upon committing a transaction, the thread tries to acquire all locks associated with the objects belonging to the transaction's write-set, as in the first phase of the two-phase locking (2PL) scheme. If at least one lock acquisition fails due to a conflict with a lock already acquired by another thread, the transaction is aborted and restarted. The transaction's read-set is validated if all locks are successfully acquired. Specifically, for each object in the read-set, the timestamp is compared with the value of *tst* to check whether the object has been modified by a concurrent transaction after the transaction started. Indeed, if an object has been modified, then its timestamp has to be greater than *tst* since it has been updated by the concurrent transaction that was committed after the transaction started. Hence, if the timestamp of at least one object in the read-set is greater than *tst*, then the transaction is aborted and restarted. Otherwise, the transaction can be successfully committed. Thus, the values stored in the write-set buffers are stored in the objects to be made visible to the other threads. Finally, all acquired locks are released.

If a transaction is aborted during the commit operation, all the acquired locks (if any) are immediately released. We note that for a read-only transaction, the commit operation is unnecessary. In fact, if no transactional write operation has to be executed prior to committing, then it is sufficient to ensure that the transaction, along its execution, has read valid object values. This is verified by the read validation executed upon each read operation.

A mechanism that can be used in combination with the presented concurrency control algorithm is *snapshot extension*. Upon a transactional read operation, if the read validation fails, the thread checks whether all the objects in the transaction read-set (if any) are still valid. In the positive case, it means that all object values read by the previous transactional read operations of the transaction are still consistent with the current transactional read operation. Consequently, abort is not required. In this case, the validity snapshot of the transaction can be extended, which consists of updating *tst* with the current *gvc* value; thus, the transaction can continue to be executed.

3.2 | Identifying a Restoration Point

As discussed in Section 1, partial abort allows avoiding squashing the whole work executed by a transaction that has to be aborted. Thus, the first issue to deal with when designing a partial abort scheme is how to identify the correct transaction restoration point, i.e. the execution point of the transaction from which the transaction has to be restarted. We remark that once a correct restoration point is identified, only modifications to data performed by the transaction after the restoration point have to be undone.

Our partial abort scheme works in synergy with read validations performed by the concurrency control algorithm to identify the restoration point. Also, it uses a read-set validation scheme similar to snapshot extension. In detail, when a transactional read operation is executed and the read validation fails, our partial abort scheme tries to validate all objects in the read-set in the same sequence the transaction has accessed them. If all objects are still valid (i.e. they have not been updated by concurrent transactions after the begin of the transaction), then the validity snapshot of the transaction can be extended, precisely like in the case of snapshot extension, thus avoiding the transaction abort. Otherwise, the first invalid object found in the read-set determines the restoration point. In other words, the restoration point is the transactional read operation that accessed the first invalid object. This ensures that all the objects read by the transactional read operations prior to the restoration point are still valid; thus, only write operations executed after the restoration point may have written inconsistent values. Consequently, only these write operations require to be undone.

3.3 | Undoing transactional work

To undo all the modifications to data performed by a transaction after a restoration point, we need to keep track of the write operations executed after that point. Given that it is impossible to know in advance which will be the selected restoration point, all the write operations executed since the start of the transaction should be tracked. As discussed, we targeted a scheme design that can undo modifications to both shared data and thread-private data made by a transaction. To this aim, we need to track: 1) all modifications made by transactional write operations to data in the shared memory, and 2) all modifications made by (non-transactional) write operations to data stored in the stack and heap memory areas of the thread which executes the transaction. Further, we need to keep track of the previous values of data modified by the transaction in order to be able to restore them. Finally, we also have to track, for any potential restoration point, the subset of modifications made by the transaction after that point. This is required for undoing only the correct portion of work made by the transaction after the point. All these aspects represent challenging issues to deal with when implementing a partial abort schema on off-the-shelf computer architectures, given the absence of specific support for tracking memory updates on the different memory areas.

Since transactional read operations represent the potential restoration points, our partial abort scheme tracks causality relations (i.e. temporal ordering) between transactional read operations and all write operations (i.e. transactional and non-transactional ones) of a transaction. In order to track causality relations between transactional read operations and transactional write operations, our scheme creates a pointer in the read-set entry of each transactional read operation. It points to the entry in the write set of the first transactional write operation executed by the transaction after the transactional read operation. Accordingly, the pointed transactional write operation and all the other transactional write operations stored in the write-set after the pointed one are causally dependent on the read operation. This scheme allows rapidly identifying the subset of transactional write operations to undo at each restoration point.

```
1  // global shared variables
2  int shared_int_a = ...;
3  int shared_int_b = ...;
4
5  // function executed by the thread
6  thread_function(int x) {
7     int a, b;
8     b = ...
9     TM_begin(); // transaction starts here
10    ...
11    a = TM_read(shared_int_a);
12    b = b + x;
13    TM_write(shared_int_b, a+b);
14    ...
15    TM_end(); // transaction ends here
16 }
```

```
1  // global shared variables
2  int shared_int_a = ...;
3  int shared_int_b = ...;
4  int shared_int_c = ...;
5
6  function1() {
7     int z = ...;
8     z += TM_read(shared_int_c);
9     TM_write(shared_int_c, z);
10    ...
11    return z;
12 }
13
14 function2(int *w) {
15    *w++;
16 }
17
18 // function executed by the thread
19 thread_function(int x) {
20    int a,b;
21    b = ...
22    TM_begin();
23    ...
24    a = TM_read(shared_int_a);
25    b = function1();
26    function2(&x);
27    TM_write(shared_int_b, a + b + x);
28    ...
29    TM_end();
30 }
```

FIGURE 1 Examples excluding (left) and including (right) application-routine calls within the transactional code block.

3.3.1 | Dealing with the stack

Tracking modifications to thread-private data is not trivial, given that any read/write operation on these data is not tracked in the read/write-set. Moreover, multiple (nested) functions might be called within a transaction, thus creating multiple stack frames, each of which can keep any number of local variables that should be tracked. We provide an example through the code listing in Figure 1. `TM_begin()` and `TM_end()` denote the start and the end of the transaction, respectively. `TM_read(var)` denotes a transactional read operation on variable `var`. `TM_write(var, new_val)` denotes a transactional write operation on variable `var`, where `new_val` is the value to be written. On the left-hand side of Figure 1, we show an example of a transaction executed by a thread. Assume that a conflict occurs after the operation `TM_write` (i.e. after code line 12), and that the restoration point is the previous `TM_read` operation (code line 10). In this case, modifications to undo include: 1) the update of the thread-private variable `a` (code line 10), 2) the update of the thread-private variable `b` (code line 11), and 3) the update of the shared variable `shared_int_c` executed by the transactional write operation `TM_write(shared_int_b, a+b)` (code line 12). We note that, while the variable `shared_int_b` is located in a shared memory area, variables `a` and `b` are in the thread stack area. Now we consider the code on the right-hand side of Figure 1. In this case, the thread also issues a call to other functions during the transaction execution. If a conflict occurs after the operation `TM_write` in `function1` (i.e. after code line 9 and before code line 11), and the restoration point is the previous `TM_read` operation in `function1` (code line 8), we need to undo the update of the local variable `z` (code line 8), which is stored in the stack of `function1`. If a conflict occurs after the operation `TM_write` of `thread_function` (i.e. after code line 26), and the restoration point is the previous `TM_read` operation (code line 23), we need to undo various updates, in particular including the update to the variable `x` executed when calling `function2` by means of dereferenced pointer. We note that any generic function might be called both within a transaction and outside of a transaction, thus making it even more complex to track updates to local variables.

To make a partial abort scheme effective, we need to correctly identify any update to any local variable used when running any portion of code of the application in a transaction, and we need to do that efficiently. Also, we should avoid paying the costs related to tracking updates to local variables whenever some portion of code is not run within a transaction.

To cope with the issues mentioned above, the partial abort scheme we designed uses an approach that restores the stack to a valid snapshot at the time in which the code line at the restoration point has been executed the last time, as depicted in Figure 2. It works as follows:

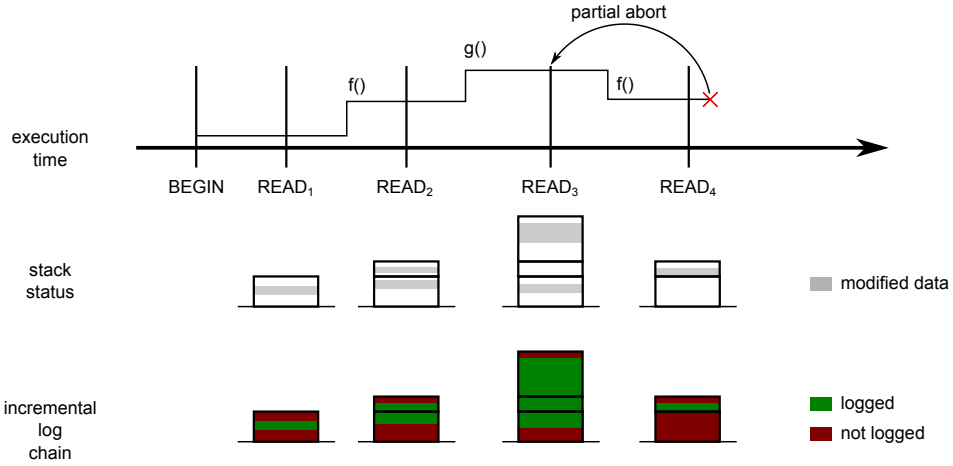

FIGURE 2 Stack management within partial rollback.

A1: when a transaction starts, the base address of the stack frame of the currently-running function is saved. As we shall describe later, this value is used while the transaction is being executed to grow an *incremental stack window*.

A2: upon the invocation of a `TM_read` operation, a recovery image for the whole memory segment in between the current stack pointer and the base pointer (whose value was previously logged, as in **A1**) is created, together with a recovery image for the processor context.

A3: when a snapshot extension operation is carried out, the stack/processor recovery image associated with the first no longer valid read operation (as generated in **A2**) is restored.

A4: upon the successful termination of `TM_end`, which commits the transaction, the recovery images associated with the transaction execution path (as determined by **A2**) are discarded.

Concerning point **A1**, we note that in modern x86_64 architecture, it is not safe to consider any memory address below the stack pointer as being part of the current thread's stack. Indeed, the modern memory layout is such that different threads' stacks can be located one after the other in the address space, and they can also be interleaved with portions of the heap. This latter point is critical, because if a thread shares a heap variable with another thread, simply checking if an address falls below the stack pointer might incorrectly deem that address as belonging to the current thread's stack. To cope with this issue, upon thread creation, we store the initial stack pointer. This value is additionally used to determine whether an address falls out of the bottom of a thread's stack.

Point **A3** provides the facilities to consistently restore an intermediate snapshot of thread-private data even if all the transactional work is squashed, e.g. due to invalidating the object accessed upon the first read operation along the transactional code block. This is a relevant facility to simplify the development of the application code.

To correctly identify the data which have to be logged in point **A2**, and also to reduce the amount of data that is copied into the log, we rely on an incremental stack window. In particular, we adopt an incremental logging strategy, based on static binary instrumentation[2]. Specifically, when the application is being linked, the produced image file is parsed in order to insert a code block before any memory write operation (e.g. `mov` instructions) which, by analyzing the current state of the processor, determines the actual virtual address to be targeted by the update, and the size of the touched memory area. If the memory-write operation falls within the stack (namely, the lower address of the area to be touched is not less than the current stack pointer value[3]), it means that the operation deals with the content of the stack thus needs to be made undoable. Hence, it is determined the interval of virtual addresses $[I_1, I_2]$ involved in the update, which in turn identifies a stack portion to be logged upon the creation of the next recovery image as the one containing all the memory locations in between the addresses I_1 and I_2. We note that for some memory instructions, write access to the stack occurs by default, such as for the case of `push` and `call`. As for

[2]For the sake of simplicity, we discuss the approach by focusing on x86-compliant architectures, but it can be easily adapted to any instruction set.

[3]To deal with the *red zone*, i.e. a fixed-size area in a function's stack frame below the current stack pointer that is reserved and safe to use by leaf functions, we simply consider an offset of 128 bytes from the top of the stack.

the `call` instruction, instrumentation determines whereinto the stack this instruction will write the return value for the program counter, while for `push`, it determines where a generic value will be written.

Multiple write operations can occur before the point where the creation of the stack recovery image occurs. For example, an additional write may involve the stack portion in the interval of addresses $[I_3, I_4]$. In such a case, instead of explicitly maintaining a list of stack areas to be logged and restored, we adopt a clustering approach where we identify the actual area to be logged as the one in between a minimum address value computed as $I^- = \min(I_1, I_3)$ and a maximum address value computed as $I^+ = \max(I_2, I_4)$. In other words, we always log a contiguous segment of the stack, which is ensured to contain all the modified stack locations (although possibly containing also non-modified locations), which is done in order to perform the actual log operation by using a single machine instruction, namely the `movs` instruction of the *x86* instruction set.

We emphasize that our approach based on a boundary check on the actually-modified region of the stack well copes with common optimizations offered by modern compiling toolchains. In particular, when high optimization levels are requested, standard compilers might decide to use, where available, the stack base registers (e.g. `ebx` on x86 architectures) as general-purpose ones. This decision allows to speed up the program's execution by enlarging the information set that the processing unit can maintain within its internal state. On the other hand, this makes it impossible to determine the current function's stack frame. Our solution can also cope with this scenario since we do not need to explicitly know the base of the stack zone for any specific function.

With the devised approach, we can undo also update operations occurring within the stack via pointer-based access. Specifically, whenever any function is activated within the execution flow of a function executed by a thread (e.g. `thread_function` in Figure 1), if any pointer is received in input which allows accessing memory locations on stack frames of other functions living along the thread, then any write access is automatically handled via the recovery scheme depicted above.

3.3.2 | Dealing with the heap

The other aspect to consider is related to updates occurring within global data that are inherently outside the control of the STM layer (e.g. non-transactional global per-thread variables stored in the heap), possibly accessed directly via pointers. This scenario is similar in spirit to the discussion which we have carried out for variables stored on the stack, yet it is simply not possible to put in place a logging strategy similar to the one discussed above. Indeed, when dealing with per-thread variables stored in the heap, two main problems arise due to the heap being shared across the threads of the same process. In this sense, although the data are accessed in data separation, they can be significantly scattered around the virtual address space. We note, however, that typical transactional code does not heavily rely on heap data.

To effectively overcome these issues when implementing a partial abort strategy targeting the heap, we suggest relying on *per-thread* allocators. In particular, each thread can rely on a memory map manager which, upon the invocation of an allocation service such as `malloc()` or `new`, retrieves a buffer from a per-thread area.

A graphical representation of the organization is depicted in Figure 3. Each thread is associated with an *allocation pool*, a compact data structure keeping a pointer to a second-level *memory pool* used to serve memory requests. The allocation pool is organized to associate each entry with a given size of memory requests to be served, only accounting for power of two sizes. In this way, upon a memory request of size m, the bucket from which to access the second-level memory pool can be quickly identified as $1U << (\log_2(m-1)+1)$—this corresponds to a fast routine composed only of a handful of instructions on most computer architectures, not really requiring to compute the costly $\log_2$ operation, being the buffers size powers of two.

Each memory pool can serve memory requests from chunks of the same size. This is a buffer of contiguous memory chunks lazily allocated from the heap. Therefore, a memory pool is allocated only after the first request of size m is received from the transactional allocation. Memory pools are augmented with a couple of bitmaps, with each bit being associated with one memory chunk. A *use bitmap* is used to keep track of memory chunks already allocated via previous `malloc`/`new` calls. Therefore, a fast bitmap scan allows identifying a free chunk that can be returned to serve a new memory request. A *dirty bitmap* can be used to keep track of what memory chunks have been accessed in write mode during the execution of a transaction, according to the following protocol:

B1: When a transaction is started, the allocated memory pools of the thread running the transaction are scanned. All the dirty bitmaps are cleared, meaning that no chunk, in the current incarnation of the transaction, has been accessed in write mode during the transaction execution. Additionally, a snapshot of all the used bitmaps is taken.

B2: When a memory-write instruction is detected to write on a memory chunk associated with the current thread, the dirty bitmap of the memory pool keeping the chunk is scanned. The corresponding bit is set to maintain the information that

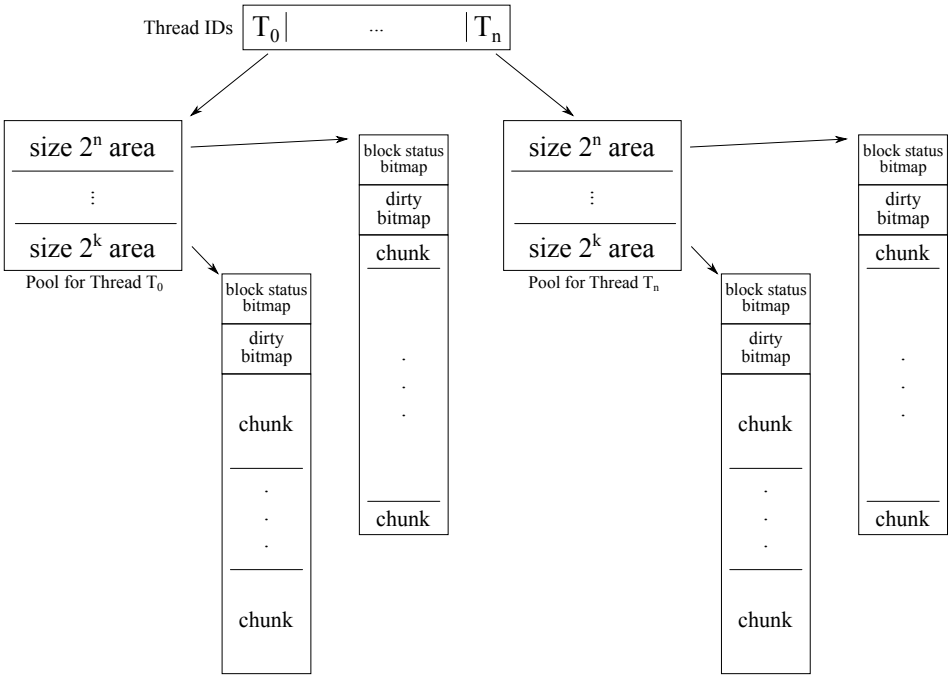

FIGURE 3 Conceptual organization of the heap.

a change in that chunk should be considered when performing a partial abort operation, and a snapshot of the previous content of the corresponding chunk is taken.

B3: Upon the invocation of a `TM_read` operation, all chunks associated with a set dirty bit belonging to all memory pools are copied in a compact data structure (a *heap snapshot*) which is linked to the transactional read-set. Moreover, also dirty bitmaps and use bitmaps are stored. The former is used to allow a fast restore of memory chunks upon a partial abort operation. In contrast, the latter is used to account for the recoverability of allocation/deallocation operations, as we shall describe.

B4: When a snapshot extension operation is carried out, all heap snapshots are scanned up to the one associated with the first no longer valid read operation. All chunks which were updated are copied back in place to the live image of the memory pools, thanks to the dirty bitmaps, which allow determining the actual memory locations from the heap snapshots. The last use bitmap is put back in place to undo the execution of possible `free`/`delete` operations executed during the transaction.

A4: Upon the successful invocation of `TM_end`, which determines the actual commitment, all heap snapshots are discarded.

Points **B3/B4** deserve an additional discussion. For the sake of generality of our approach, it might be possible that while executing a transaction, a thread might allocate/release memory buffers. In case of a partial abort, these actions should also be undone to restore a consistent memory snapshot from which to restart the transaction's execution. Let us consider the execution depicted in Figure 4. A transaction is started after that a buffer of size *m* was acquired from the heap, namely from the thread pool. This buffer is freed during the transaction execution, therefore clearing the associated use bit. A subsequent malloc operation of the same size *m* will likely return a pointer to the same memory chunk. Upon a partial abort operation, the restoration point is identified as immediately before the `free()` operation, which released the buffer of size *m*. Also, in this scenario, the above-described protocol brings back the memory snapshot to a correct state thanks to point **B4** because: i) the use bitmap associated with the state before the `free()` operation (saved as per point **B3**) is put back in place, telling that the memory chunk pointed by `ptr1` is valid, and ii) the content of the previous incarnation of the memory buffer is written back in place, taking it from heap snapshots or the initial snapshot (taken at point **B1**).

The overall protocol to manage the heap might appear costly. Anyhow, we emphasize again that the performance is not the only goal of our partial abort scheme, but it also aims at providing full transparency to the programmer, in particular when writing transactional code that access non-transactional data. In Section 5, we will experimentally assess the overhead of the

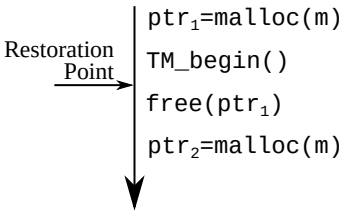

FIGURE 4 Sequence of `malloc`/`free` operations.

proposed protocol, showing anyhow that for a reduced amount of memory operations within a transaction, this can be considered negligible.

4 | IMPLEMENTATION DETAILS

The specific implementation we provide of the partial rollback scheme devised in Section 3.2 has been tailored for integration within TinySTM, an open-source `C` based STM layer widely diffused in academic contexts, which is typically adopted for prototypical development and validation of innovative research results. The implementation has been based on hijacker[46], a general-purpose static binary instrumentation tool targeting differentiated software manipulation requirements and objectives. Hijacker allows altering (at linking time) the actual operations performed by an executable without modifying its semantics. We have targeted Executable and Linkable Format (ELF) objects generated by standard GCC compilers on *x86-64* architectures. Additionally, hijacker allows generating multiple versions of the same code (accessing the same data), which is manipulated in differentiated ways, depending on the instrumentation needs. This technique, known as *multi-coding*, allows putting in place more complex instrumentation mechanisms, still reducing the possible overhead at runtime. By specifying a set of XML-based rules, we are able to alter transactional code by:

- Identifying all memory-write instructions within code blocks explicitly marked as transactions;
- Prepend to them a call to ad-hoc routines which allow tracking memory updates according to the protocols described in Section 3;
- Generate copies of all nested functions, called from a transactional context, and instrument them according to the same scheme[4];
- Replace all function calls within a transactional context, targeting the instrumented copies.

The additional version of functions instrumented only for the transactional context reduces the runtime overhead. Indeed, if the same function `f()` is called from a non-transactional context, the original version will be executed. On the other hand, from a transactional context, the flow is diverted to `f'()`, which is a function with the same semantic of `f()`, yet it is instrumented to track memory updates. In this way, thread execution along a non-transactional context will not pay the overhead associated with static binary instrumentation. All these modifications allow for complete transparent instrumentation of the STM-based application, enabling partial abort without manual intervention by the programmer.

Similarly, TinySTM has been augmented by introducing the logic required to handle partial abort operations, namely:

- Identifying the execution points where recovery images for the stack of the transactional thread need to be taken;
- Implementing the actual log/restore logic for the stack and its management combined with the management of read-/write-sets.

Both the above points have been achieved by nesting a block of code to be executed right before any call by the application software to the `TM_read` function offered by the TinySTM API. This allows to transparently take control right upon the need

[4]We assume that all call sites can be identified at inspection time. This leaves out the possibility to call functions via pointers. To support function pointers, more costly operations should be supported, namely instrumenting all functions and determining at runtime whether we are running in a transactional context or not. For the sake of performance, we have explicitly discarded this option.

```
stack_pointer = stack_pointer(); // Read RSP
getcontext(&cpu_state); // cpu_state is a jmp_buf
stm_store_context_in_readset(&cpu_state);
```

FIGURE 5 Instructions prepended to `TM_read`.

```
// C-equivalent declarations
   typedef struct {
       void *upper;
       void *lower;
   } stack_region_t;

   __thread stack_region_t stack_region;

// Used assembly code
  .tbss
   stack_region: .zero 16

  .text

   push  %rax
   push  %rdi
   lea   variable, %rdi
   movq  %fs:stack_region@tpoff, %rax
   testq %rax, %rax
   je    3f
   cmpq  %rdi, %rax
   jbe   2f
1: cmpq  %fs:stack_region@tpoff+8, %rdi
   ja    4f
   movq  %rdi, %fs:stack_region@tpoff+8
   jmp   4f
2: movq  %rdi, %fs:stack_region@tpoff
   jmp   1b
3: movq  %rdi, %fs:stack_region@tpoff+8
   movq  %rdi, %fs:stack_region@tpoff
4: pop   %rdi
   pop   %rax
```

FIGURE 6 Identification of the stack region to be saved.

for creating a stack recovery image, namely before accessing the target transactional object in read mode. If the read operation is revealed as invalid, the additional logic included within the TinySTM layer is exploited to exploit stack restoration images for supporting partial rollback.

In order to correctly create a recovery image, which includes the current processor context, we have prepended to the operations by the `TM_read` function the instructions reported in Figure 5. With this approach, stack/processor information associated with the current state of execution of the function calling `TM_read` (namely `thread_function` in the example in Figure 1) is sampled, with no modification of stack pointer/content and CPU image performed by the code block. On the other hand, the actual creation of the stack recovery image is demanded from internal logic we have added within TinySTM, which performs optimized management of stack log operations, as we shall explain in what follows. This is accessed via the `stm_store_context_in_readset` API, which we have added to the internals of TinySTM.

The identification of the stack region to be saved for future re-installation upon a partial abort operation is based on the injection, within the instrumented code, of a snippet as shown in Figure 6. It implements the boundary check described in Section 3.3.1 in a fast way. In particular, a per-thread struct kept in the TLS is used to keep track of the current boundaries I^- and I^+. These values (which are cleared upon a transaction startup) are checked against the current address of the variable being accessed[5]. If the address falls out of the current range $[I^-, I^+]$, the proper upper or lower limit is updated.

[5]For simplicity, we call the accessed location `variable` in the code snippet. Thanks to the `lea` instruction, more complex addressing methods can be evaluated with the same single operation.

As mentioned, upon the detection of an inconsistent value read during the transaction's execution, instead of relying on the classical rollback scheme, we perform a partial rollback operation, which entails restarting the execution of the conflicting transaction from an intermediate point such that every operation before it is still considered valid.

In order to effectively restart from within a transaction, we must restore every aspect of the execution context. If, on the one hand, the processor state is restored via the standard System V `setcontext` library function—using a previously stored snapshot—in order to cope with automatic variables successfully, we must undo any modification concerning the stack frames of the functions called during the transaction.

Concerning heap management, a couple of actions should be taken to transparently inject in the transactional application the heap manager described in Section 3.3.2. In particular:

- All calls to allocation/deallocation services (such as `malloc()`/`free()`) should be intercepted and redirected to the custom heap manager implementation.
- Library calls that spawn new threads should similarly be wrapped to setup the ad-hoc memory manager.

Both points above are achieved thanks to standard linker wrapping tools. In particular, the GNU linker `ld` provides the `-wrap` flag, which allows specifying a target library function to be redirected to a custom wrapper which, in turn, can have access to the original implementation. We have specifically targeted `malloc()`, `free()`, and `pthread_create()` library functions. The former two have been replaced with the custom implementation of our heap manager. The latter has been used to initialize the data structures associated with the heap manager before calling into the original thread activation function.

The initialization of the heap manager relies on a per-thread variable stored in the TSS of the thread, which points to the memory pool of each thread. In this way, every time that a call to the intercepted `malloc()`/`free()` functions is issued, the running thread can access the correct per-thread pool. This solution also allows to cope with transactional applications that use any number of threads in a straightforward way.

5 | EXPERIMENTAL STUDY

As mentioned, we implemented our partial abort scheme within the open-source TinySTM library[3] for evaluation purposes. In this section, we present the results of an evaluation study we conducted for assessing its impact on performance. We used the following five benchmark applications from the well-known STAMP benchmark suite[47] for STM:

1. bayes is a transactional implementation of an algorithm to learn Bayesian networks. A Bayesian network represents a probability distribution for a set of variables in a graphical manner. It is a directly acyclic graph where each node represents a variable and each edge represents a conditional dependence. bayes implements a hill-climbing algorithm employing both a local and a global search[48]. This benchmark relies heavily on dynamic memory, therefore it is a good benchmark to study the effects of dynamic memory management. At the same time, the transactions have a medium granularity, with a reduced amount of read operations, thus serving as an average case with respect to overhead/performance benefit to assess our approach.

2. ssca2 is a transactional implementation of the Scalable Synthetic Compact Applications 2 (SSCA2) benchmark[49], where a graph kernel is used to build a directed, weighted multi-graph using adjacency arrays and auxiliary arrays. In particular, threads concurrently add nodes to the graph, and transactions are used to synchronize accesses to the adjacency arrays. This kind of operation is relatively fast, therefore the transactions' granularity and the amount of data involved is relatively small. This entails a reduced data contention, making this benchmark effective for assessing the actual overhead produced by our partial rollback implementation with respect to the traditional rollback scheme.

3. genome implements a transactional algorithm to reconstruct a gene sequence from segments of a larger gene. This benchmark relies on multiple algorithms that have a non-minimal impact on transaction duration. In particular, it relies on a hash set to identify duplicated segments, and it relies on the Rabin-Karp string search algorithm[50] to match segments. As these operations are parallelized, the amount of concurrent read/write operations is non-minimal, especially in case of large datasets.

4. kmeans is a transactional implementation of a partition-based clustering method[51]. A cluster is represented by the mean value of all objects that it contains, and during the execution of this benchmark, the mean points are updated by assigning

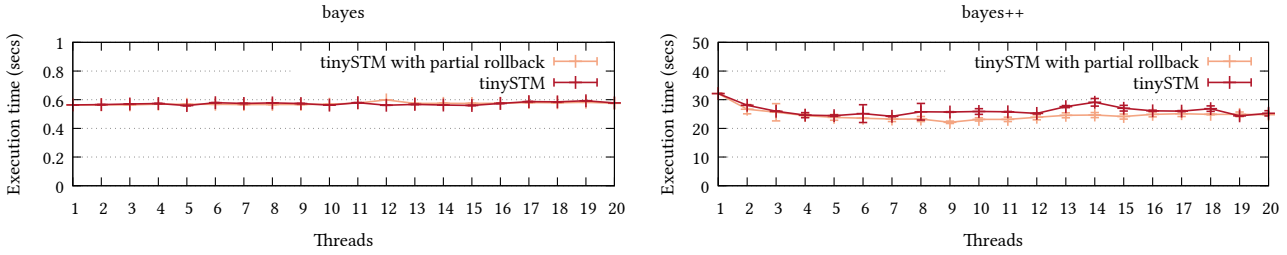

FIGURE 7 Results with bayes.

each object to its nearest cluster center, based on Euclid distance. This particular benchmark relies on threads working on separate subsets of the data and uses transactions to both assign portions of the workload and store final results concerning the new centroid updates. Although the granularity of transactions is relatively small in this benchmark as well, given the reduced amount of shared data structures being updated, it is more likely to incur in logical contention when a larger number of threads is used to follow through the computation, thus giving us the possibility to show the benefits deriving from our partial rollback scheme. In particular, we note that this is not the best case for our approach since the amount of saved work is nevertheless reduced, so that the overhead generated by the CPU/stack state saving/restoring could not be completely amortized, which gives rise to a meaningful test case.

5. vacation is an application implementing a travel reservation system. It is powered by a single-instance database, and the clients interact with it by means of a transaction manager. Clients can issue four different requests, i.e. to make a reservation, to delete a customer, to insert a car, flight, or room in the database, or to remove such elements. Tables in the database are implemented as Red-Black trees. Similarly to bayes, this benchmark heavily uses dynamic memory, while it has transactions that are typically short.

According to the STAMP specification, all the selected benchmark applications can be characterized by two parameters. One is the size of the data set, which can be either small, medium (marked with '+'), and large (marked with '++'). The second one, mainly used for the `kmeans` and `vacation` benchmarks, indicates the actual requirements of the transactions, in terms of, e.g., the actual span of the accesses onto the data set and, correspondingly, CPU requirements. It is also an indication of the relative amount of contention. This parameter is denoted as 'high' (indicating high demand) and 'low' (indicating reduced demand).

The computing platform used in the experiments is equipped with a 3.70GHz Intel(R) Core(TM) i9-10900X CPU, with 10 cores (20 hyper threads) and 16 GB of RAM. The used operating system is Ubuntu Server 22.04 LTS, with Linux kernel 5.15.0. The used compiler is clang 14.0.0. All the data provided in this section are averaged over 30 different runs, and report results when varying the number of concurrent threads from 1 to 20. We also show the estimated 90% confidence intervals.

In Figure 7 we report the execution time for the bayes benchmark, which is characterized by long transactions and generally high contention. In this particular benchmark, the partial rollback approach has to create a non-minimal number of snapshots, increasing the overhead. Also, snapshot management is such that, upon a partial rollback operation, unneeded snapshots have to be garbage collected. This particular scenario makes the approach unable to payoff the cumulative overhead of a long transaction. Nevertheless, when the data set is larger (the '++' configuration), the partial rollback scheme is able, at high contention levels, to deliver some performance improvement, although marginal. This is related to the reduced pressure on the housekeeping operations.

The results for `scca2` are shown in Figure 8. As we already mentioned before, this benchmark is characterized by short transactions, accessing a reduced amount of data, and the time spent in transactions is a reduced percentage of the whole application execution time. In the study presented in Minh et al.[49], it has been estimated to be at most 17%, and the abort probability is about 0.01. The results in Figure 8 show a minimal overhead induced by our scheme. Specifically, for the case of a small dataset configuration, we observe an overhead on the order of 7% for most of the considered concurrency levels (i.e. number of threads). Such overhead is further reduced when considering the '++' configuration, where the increased data set leads to secondary effects, such as those related to the reduced locality, to be more evident, with a consequent reduction of the relative overhead for the CPU context/stack logging mechanisms injected within TinySTM via our solution.

In Figure 9 we report the execution time for the genome benchmark when varying the number of concurrent threads in between 1 and 20. As we can see from the results, the partial rollback approach produces no performance improvement. Nevertheless, we

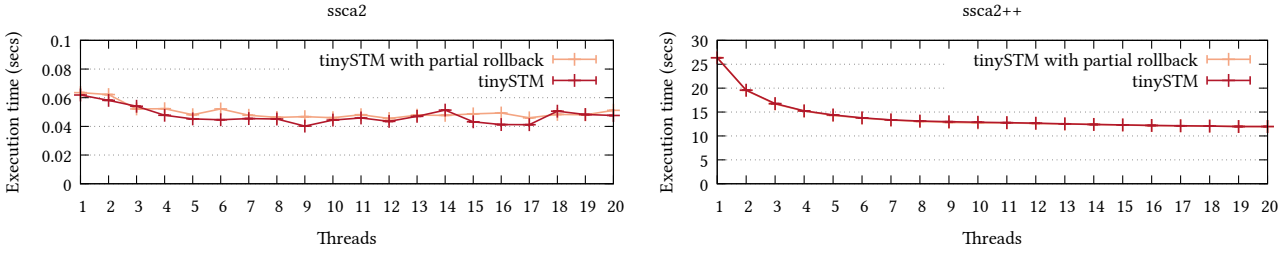

FIGURE 8 Results with scca2.

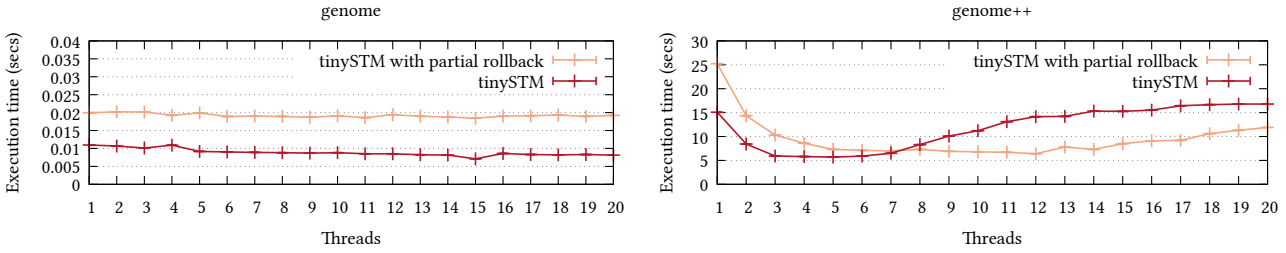

FIGURE 9 Results with genome.

observe that with an increased amount of data to be processed transactionally, at higher thread counts the performance penalty paid by our approach is reduced. This is an additional indication that resuming a transaction execution from the last valid read can pay off the overhead required by the procedures needed to reconstruct both the memory and the CPU states.

In Figure 10, we report the results we observed with `kmeans`. With `kmeans`, the most unfavorable configuration for our partial abort scheme is 'high+', where the transactions have higher requirements and access a smaller data set. This leads to scenarios of high data conflict, likely occurring in the early phase of transaction execution (due to the reduced size of the data set). This leads the partial abort scheme to induce a non-minimal amount of logging overhead while not allowing a sufficient save of work while undoing transactions. We noticed that this is because most transactions were restarted from the beginning. This phenomenon is alleviated when considering the 'high++' configuration, where the increased size of the data set leads to scenarios where at least a portion of the performed transactional work can be successfully saved since the data set largeness reduces the likelihood of conflicts in the early phase of transaction execution. This leads the partial abort scheme to exhibit increased effectiveness, especially when concurrency is higher. In such a case, in fact, it achieves up to a 20% reduction of the application execution time in the case of 20 CPU cores.

The 'low+' configuration of kmeans gives rise to execution dynamics that are not so far from those observable for the 'high++' configuration. Specifically, transactions conflict while accessing a reduced data set, but they exhibit low resource requirements. Hence, also in this case, there is no bias towards conflicting in the early execution phase. Consequently, the scheme operates effectively, especially when the level of concurrency is increased. Indeed, it provides a reduction of the application execution time in the order of 40% as soon as the number of used CPU cores is greater than 23.

On the other hand, with reduced concurrency levels, the impact of transaction abort is reduced, which leads to the scenario where the partial abort scheme induces a logging overhead that does not get compensated by revenues while partially undoing transactions. This overhead is further absorbed when running the 'low++' configuration (e.g., due to the aforementioned reduced locality phenomena within the benchmark). Hence for this scenario, we observe that our scheme provides a similar performance achievable by fully-aborted transactions when the level of concurrency is limited. Conversely, it provides some performance advantages when this level increases, leading to increased abort probability, thus making the partial save of transactional work already carried out more useful.

Similar conclusions can be drawn from the vacation benchmark, whose results are provided in Figure 11. Again, with low contention, the overhead paid by the partial rollback scheme is not paid off, while it is the case when the contention degree is increased and the number of concurrent threads grows. Interestingly, from the results we also observe that when the contention

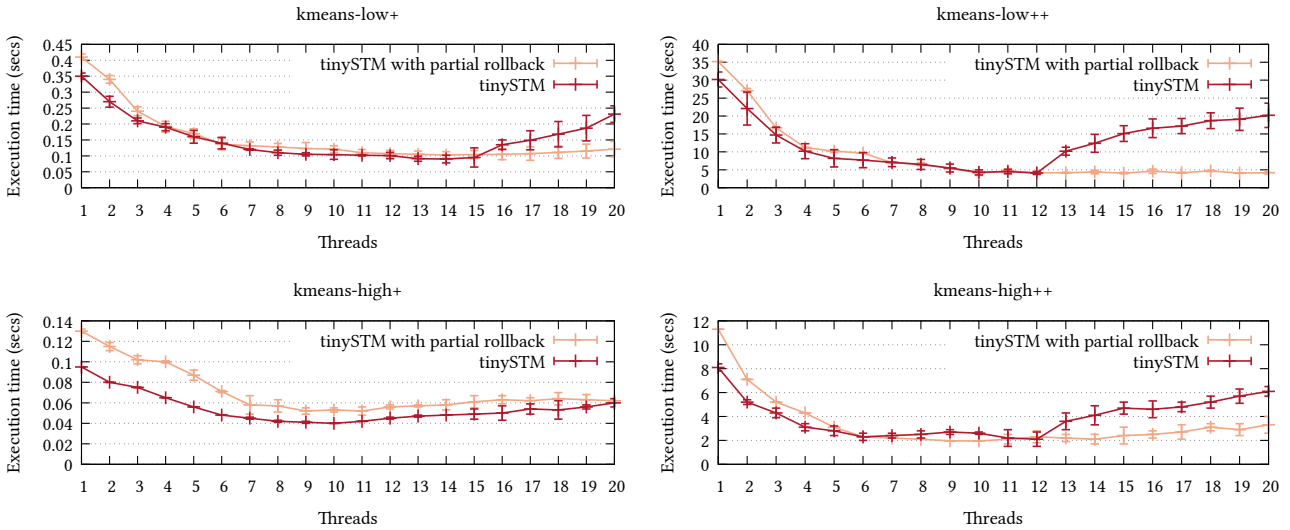

FIGURE 10 Results with kmeans.

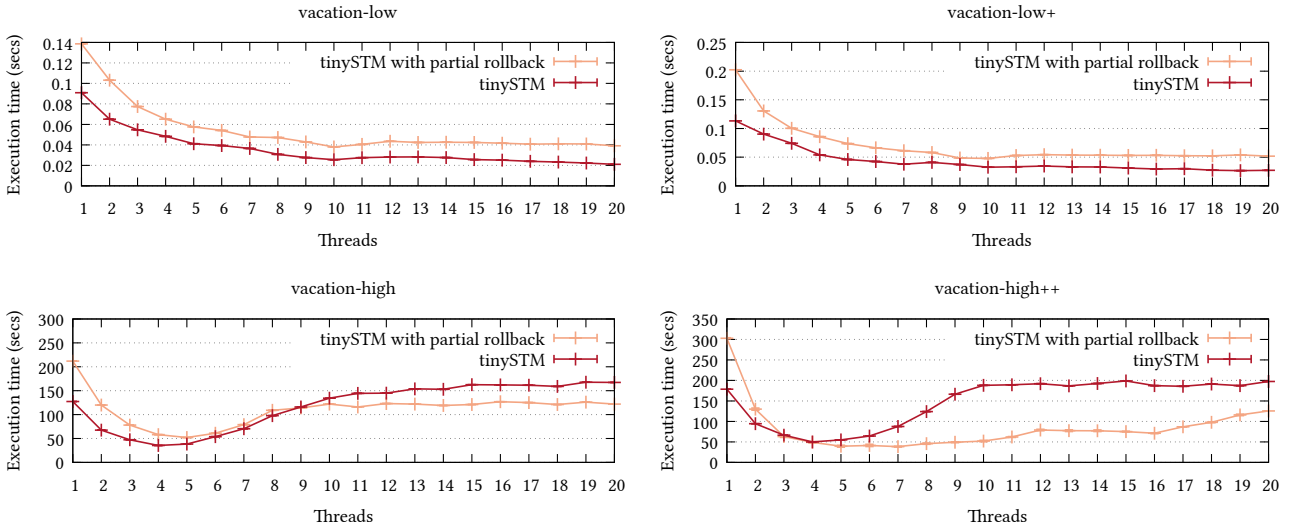

FIGURE 11 Results with vacation.

increases, but over a larger dataset, the benefit of the partial rollback scheme becomes more apparent—this is particularly evident in the vacation-high++ configuration.

To complete the experimental assessment, we provide in Figure 12 the time spent in the application benchmarks to manage the stack of the different threads. The plot refers to the configuration run with 20 concurrent threads, and shows the cumulative time spent in stack management operation for the whole experiment run. By the results, we observe that the time spent in the management of the stack is negligible compared to the duration of the overall benchmark. In particular, in all configurations it is always lower than 10% and, in most cases, below 5%.

For a more accurate assessment of our partial abort scheme's heap management strategy, we use a different application. In effect, available benchmarks for transactional applications make less use of non-transactional accesses in the heap compared to other kinds of data accesses. This is because they focus on evaluating undo capabilities on transactional operations. Thus, we developed a specific synthetic test application that purposely allocates, for each thread, a set of data structures from the heap. These data structures have uniformly distributed sizes in the range [128B, 4KB]. The total memory allocated from the heap by

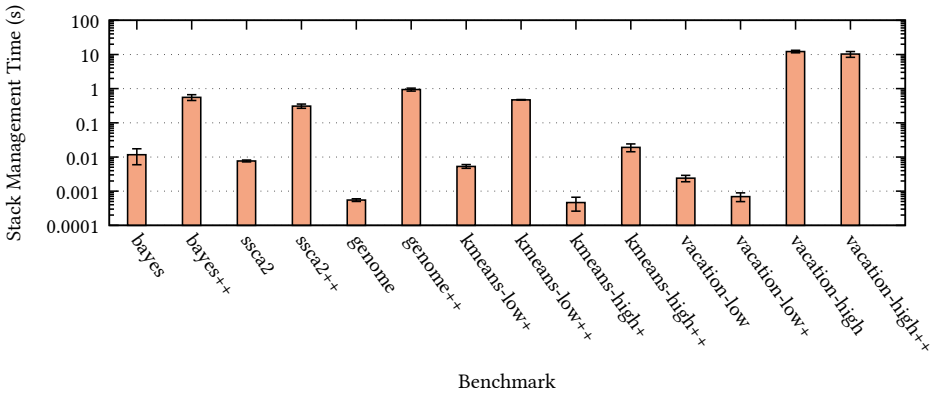

FIGURE 12 Time spent in stack management operations (20 concurrent threads).

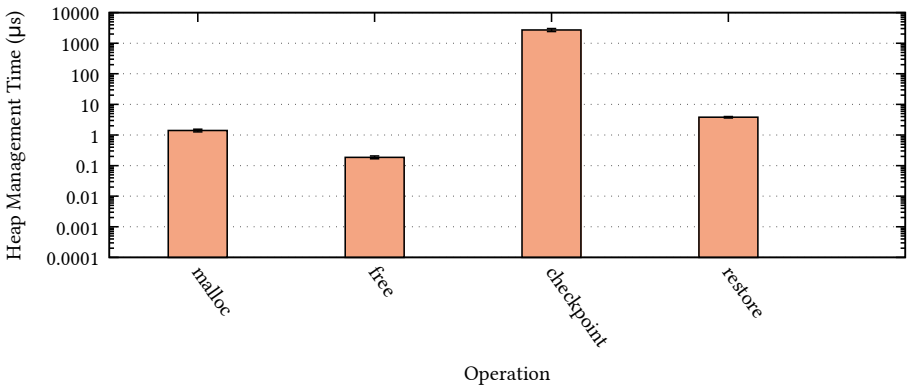

FIGURE 13 Time spent in heap management operations (20 concurrent threads).

all threads amounts to 128MB. The threads also share a small set of variables that can be accessed by transactional operations—in the order of tens. In our test, the threads perform 50 different operations during the transaction execution, randomly picked between allocation/deallocation, access in write mode to a sequence of bytes from the heap, access in read/write mode from a transactional object. Each thread executes 10,000 transactions before terminating.

As far as accesses to the heap are concerned, each thread scans every data structure allocated in the heap. With a probability of 0.5, up to 10 bytes are written in each data structure. This allows setting a non-negligible amount of dirty bits in the dirty bitmap of the memory pool of each thread, which later affects the time required to take a log and/or restore a previous snapshot. Allocation and deallocation operations are carried out to keep each thread's average amount of memory used from the heap. The access to transactional objects is carried out randomly, therefore causing possible conflicts between concurrent transactions. We have run this experiment by relying on 20 concurrent threads. The results are reported in Figure 13.

We note that this setup can be a worst-case scenario for a transactional system. To the best of our knowledge, no transactional application exhibits this pattern concerning heap accesses. Therefore, this configuration can be effectively used to assess the impact of our heap management protocol. By the results, we observe that each operation impacts the execution time in a negligible way. Indeed, the most costly operation is the creation of a checkpoint, which takes on average a time in the order of a few milliseconds. Compared to the total execution time of the benchmark (around 5 seconds), this does not significantly affect the overall execution time.

Moreover, we note that, by the design of the synthetic memory access pattern, taking a log entails creating a snapshot of around 10 MB, only to account for updates which are, on average, 32 KB. This is because the checkpoint/restore strategy we have envisaged works on a memory-chunk basis. We deal with large chunks (up to 4KB), which are subject to updates of a few bytes. From these results, we can conclude that our heap management strategy is effective, taking into account that typical transactional applications might only rely on heap updates during the transaction execution marginally.

To conclude, we remark that any performance advantage observed in specific settings is achieved in combination with application transparency of our abort scheme, which, beyond performance, was the fundamental objective of our work.

6 | FINAL ASSESSMENT, CONCLUSIONS, AND FUTURE WORK

This article presented our experience in designing and implementing a partial abort scheme for Software Transactional Memory. We remark that our objective was threefold: 1) enabling STM to perform partial abort of transactions in the case of conflicts, 2) designing a scheme fully transparent to the programmer, thus also allowing to run existing applications without modifications, and 3) implementing a scheme that can be integrated within existing STM runtime environments, running on conventional off-the-shelf architectures, without requiring to modify the STM internal implementation.

We devised a partial abort scheme that works with concurrency control based on CTL with read validation and exploits a technique similar to snapshot extension for identifying the restoration point in the case of a partial abort. Also, we achieved a scheme that does not require the programmer to write idempotent transactional code, thanks to its ability to undo both modifications to shared transactional data and thread-private data while partially aborting transactions. We successfully integrated it within the open-source TinySTM library, which we used for evaluation. We tested our implementation on top of a 20-core machine by running some applications selected from the STAMP benchmark suite and a specific application for a in-dept assessment of some components of our scheme.

In summary, by our experience, we can draw the following lesson:

- Injecting in the STM-based application at compile time instructions for enabling partial undo via static software instrumentation, like CPU-context/stack logging, introduces low overhead and has shown to work correctly. Thus, this approach can represent a key option for designing effective partial abort schemes.
- The heap management strategy we devised appears to work effectively, even with low usage of heap memory allocation. Thus, it can represent a good candidate strategy for heap management in a partial abort scheme.
- The overall effectiveness in terms of performance of our solution depends on the actual contention level and the workload profile of the transactional application. Generally, with low contention levels and/or short transaction profiles, the logging overhead could not compensate for the partial saved work of aborted transactions. Conversely, with high contention levels and longer transactions, the work performed by the partial abort scheme generally translates into effective performance advantages. This opens the way to investigate additional techniques that could improve our scheme. For example, the performance loss that arises with low conflict probability could be eliminated through dynamic activation of the partial abort scheme when the conflict probability overcomes a given threshold. Also, logging techniques that restrict as much as possible the subsets of memory areas to be logged could be explored.

Based on the results of our study, we are currently in the process of implementing and tuning our partial abort scheme to be integrated within GCC's `libtim`. A preliminary implementation available for the research community can be already found online[6].

References

1. Sutter Herb. *Welcome to the Jungle: Or, a Heterogeneous Supercomputer in Every Pocket.* : Sutter's Mill: Herb Sutter on software development; 2014.
2. Shavit Nir, Touitou Dan. Software transactional memory. In: *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press 1995.
3. Felber Pascal, Fetzer Christof, Riegel Torvald. Dynamic performance tuning of word-based software transactional memory. In: *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP. New York, NY, USA: ACM 2008 (pp. 237–246).

[6]https://github.com/HPDCS/gcc-partial-abort

4. Dragojević Aleksandar, Guerraoui Rachid, Kapalka Michal. Stretching transactional memory. *ACM SIGPLAN Notices.* 2009.

5. Herlihy Maurice, Luchangco Victor, Moir Mark, Scherer William N.. Software transactional memory for dynamic-sized data structures. In: *Proceedings of the 22nd annual Symposium on Principles of distributed computing*. PODC. ACM Press 2003 (pp. 92–101).

6. GNU Project . GCC, the GNU Compiler Collection .

7. Yoo Richard M., Hughes Christopher J., Lai Konrad, Rajwar Ravi. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. New York, New York, USA: ACM Press 2013 (pp. 1–11).

8. Ramadan Hany E., Roy Indrajit, Herlihy Maurice, Witchel Emmett. Committing conflicting transactions in an STM. *ACM SIGPLAN Notices.* 2009;44(4):163.

9. Economo Simone, Silvestri Emiliano, Di Sanzo Pierangelo, Pellegrini Alessandro, Quaglia Francesco. Prompt Application-Transparent Transaction Revalidation in Software Transactional Memory. In: *Proceedings of the 16th IEEE International Symposium on Network Computing and Applications*. NCA, vol. 2017-Janua: IEEE Computer Society 2017 (pp. 114–119).

10. Fernandes Sérgio Miguel, Cachopo João. Lock-free and scalable multi-version software transactional memory. *ACM SIGPLAN Notices.* 2011;46(8):179.

11. Natarajan Aravind, Mittal Neeraj. False conflict reduction in the Swiss Transactional Memory (SwissTM) system. In: *Proceedings of the 2010 International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IPDPSW. IEEE 2010 (pp. 1–8).

12. Felber Pascal, Fetzer C, Marlier P, Riegel T. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems.* 2010;21(12):1793–1807.

13. Ansari Mohammad, Luján Mikel, Kotselidis Christos, Jarvis Kim, Kirkham Chris, Watson Ian. Steal-on-Abort: Improving Transactional Memory Performance through Dynamic Transaction Reordering. In: Seznec A., Emer J., O'Boyle Michael F. P., Martonosi Margaret, Ungerer T., eds. *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. HiPEAC. Berlin, Heidelberg: Springer 2009 (pp. 4–18).

14. Yoo Richard M, Lee Hsien-Hsin S. Adaptive transaction scheduling for transactional memory systems. In: *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*. SPAA. ACM 2008 (pp. 169–178).

15. Ansari Mohammad, Kotselidis Christos, Jarvis Kim, Luján Mikel, Kirkham Chris, Watson Ian. Advanced Concurrency Control for Transactional Memory Using Transaction Commit Rate. In: *Proceedings of the 14th international Euro-Par conference on Parallel Processing*. Euro-Par. Springer-Verlag 2008 (pp. 719–728).

16. Dolev Shlomi, Hendler Danny, Suissa Adi. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In: *Proceedings of the 27th Symposium on Principles of Distributed Computing*. PODC. New York, New York, USA: ACM Press 2008 (pp. 125).

17. Spear Michael F, Dalessandro Luke, Marathe Virendra J, Scott Michael L. A comprehensive strategy for contention management in software transactional memory. *SIGPLAN Notices.* 2009;44(4):141–150.

18. Lev Yossi, Luchangco Victor, Marathe Virendra J, Moir Mark, Nussbaum Dan, Olszewski Marek. Anatomy of a scalable software transactional memory. In: *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*. TRANSACT. ACM 2009.

19. Rughetti Diego, Di Sanzo Pierangelo, Ciciani Bruno, Quaglia Francesco. Machine Learning-Based Self-Adjusting Concurrency in Software Transactional Memory Systems. In: *Proceedings of the 20th IEEE International Symposium On Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. MASCOTS. IEEE 2012 (pp. 278–285).

20. Rughetti Diego, Sanzo Pierangelo Di, Ciciani Bruno, Quaglia Francesco. Dynamic Feature Selection for Machine-Learning Based Concurrency Regulation in STM. In: *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE 2014 (pp. 68–75).

21. Di Sanzo Pierangelo. Analysis, Classification and Comparison of Scheduling Techniques for Software Transactional Memories. *IEEE Transactions on Parallel and Distributed Systems.* 2017;28(12):3356–3373.

22. Di Sanzo Pierangelo, Pellegrini Alessandro, Sannicandro Marco, Ciciani Bruno, Quaglia Francesco. Adaptive Model-Based Scheduling in Software Transactional Memory. *IEEE Transactions on Computers.* 2020;69(5):621–632.

23. Koskinen Eric, Herlihy Maurice P. Checkpoints and continuations instead of nested transactions. In: *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA. New York, NY, USA: ACM 2008 (pp. 160–168).

24. Adl-Tabatabai Ali-Reza, Lewis Brian T., Menon Vijay, Murphy Brian R., Saha Bratin, Shpeisman Tatiana. Compiler and runtime support for efficient software transactional memory. In: *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. PLDI. New York, New York, USA: ACM Press 2006 (pp. 26).

25. Wu Peng, Michael Maged M, Praun Christoph, et al. Compiler and runtime techniques for software transactional memory optimization. *Concurrency and computation: practice & experience.* 2009;21:7–23.

26. Xiao Yang, Jeyakumaran Thireshan, Atoofian Ehsan, Jannesari Ali. Improving performance of transactional memory through machine learning. *Concurrency and Computation: Practice and Experience.* 2018;30(10):e4397. e4397 CPE-17-0202.R2.

27. Zhou Naweiluo, Delaval Gwenaël, Robu Bogdan, Rutten Eric, Méhaut Jean-François. An autonomic-computing approach on mapping threads to multi-cores for software transactional memory. *Concurrency and Computation: Practice and Experience.* 2018;30(18):e4506. e4506 cpe.4506.

28. Koskinen Eric, Herlihy Maurice P. Checkpoints and continuations instead of nested transactions. In: *Proc. of SPAA*. ACM 2008 (pp. 160–168).

29. Moore K.E., Bobba Jayaram, Moravan M.J., Hill M.D., Wood D.A.. LogTM: Log-based Transactional Memory. In: *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*. IEEE 2006 (pp. 258–269).

30. Moravan Michelle J., Bobba Jayaram, Moore Kevin E., 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. New York, New York, USA: ACM Press 2006 (pp. 359).

31. Lupei Daniel. A Study of Conflict Detection in Software Transactional Memory. PhD thesis. University of Toronto, the Netherlands, 2009.

32. Gupta Monika, Shyamasundar Rudrapatna K, Agarwal Shivali. Article: Clustered Checkpointing and Partial Rollbacks for Reducing Conflict Costs in STMs. *International Journal of Computer Applications.* 2010;1(22):80–85.

33. Gupta Monika, Shyamasundar Rudrapatna K, Agarwal Shivali. *Automatic Checkpointing and Partial Rollback in Software Transaction Memory.* 2012.

34. Le Matthew, Fluet Matthew. Partial aborts for transactions via first-class continuations. *SIGPLAN Not.*. 2015;50(9):230–242.

35. Bestavros Azer, Braoudakis Spyridon. Value-cognizant Speculative Concurrency Control. In: *Proceedings of VLDB*. VLDB. Morgan Kaufmann Publishers Inc. 1995 (pp. 122–133).

36. Riegel Torvald, Felber Pascal, Fetzer Christof. *Distributed Computing* Lecture Notes in Computer Science, vol. 4167: . Berlin, Heidelberg: Springer Berlin Heidelberg; 2006.

37. Marathe Virendra J, Spear Michael F, Heriot Christopher, et al. Lowering the Overhead of Nonblocking Software Transactional Memory. In: *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. 2006.

38. Toccaceli Roberto, Quaglia Francesco. DyMeLoR: Dynamic Memory Logger and Restorer Library for Optimistic Simulation Objects with Generic Memory Layout. In: *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society 2008 (pp. 163–172).

39. Pellegrini Alessandro, Vitali Roberto, Quaglia Francesco. Di-DyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects. In: *Proceedings of the 23rd Workshop on Principles of Advanced and Distributed Simulation*. PADS. IEEE 2009 (pp. 45–53).

40. Quaglia Francesco, Santoro Andrea. Non-Blocking Checkpointing for Optimistic Parallel Simulation: Description and an Implementation. *IEEE Transactions on Parallel and Distributed Systems.* 2003;14(6):593–610.

41. Fujimoto Richard M, Tsai J J, Gopalakrishnan G. Design and Evaluation of the Rollback Chip: Special Purpose Hardware for Time Warp. *IEEE Transactions on Computers.* 1992;41(1):68–82.

42. Steinman Jeffrey S.. SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete Event Simulation.. *International Journal on Computer Simulation.* 1992:251–286.

43. Das Samir R, Fujimoto Richard M, Panesar Kiran, Allison Don, Hybinette Maria. GTW: a time warp system for shared memory multiprocessors. In: *WSC '94: Proceedings of the 26th conference on Winter simulation*. Society for Computer Simulation International 1994 (pp. 1332–1339).

44. Pellegrini Alessandro, Vitali Roberto, Quaglia Francesco. Autonomic State Management for Optimistic Simulation Platforms. *IEEE Transactions on Parallel and Distributed Systems.* 2015;26(6):1560–1569.

45. Dice Dave, Shalev Ori, Shavit Nir. Transactional Locking II. In: *Proceedings of the 20th International Symposium on Distributed Computing*. 2006 (pp. 194–208).

46. Pellegrini Alessandro. Hijacker: Efficient static software instrumentation with applications in high performance computing. In: *2013 International Conference on High Performance Computing & Simulation (HPCS)*. HPCS. Helsinki, Finland: IEEE 2013 (pp. 650–655).

47. Cao Minh Chi, Chung JaeWoong, Kozyrakis Christos, Olukotun Kunle. STAMP: Stanford Transactional Applications for Multi-Processing. In: *Proceedings of the IEEE International Symposium on Workload Characterization*. ISWC. 2008.

48. Chickering David Maxwell, Heckerman David, Meek Christopher. A Bayesian approach to learning Bayesian networks with local structure. In: *Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence*. UAI'97. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. 1997 (pp. 80–89).

49. Bader David A, Madduri Kamesh. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In: *Proceedings of the 12th international conference on High Performance Computing*. HiPC'05. Berlin, Heidelberg: Springer-Verlag 2005 (pp. 465–476).

50. Karp Richard M, Rabin Michael O. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development.* 1987;31(2):249–260.

51. Bezdek James C. *Pattern Recognition with Fuzzy Objective Function Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers; 1981.

How to cite this article: Alessandro Pellegrini, Pierangelo Di Sanzo, Andrea Piccione, and Francesco Quaglia (2022), Design and Implementation of a Fully-Transparent Partial Abort Support for Software Transactional Memory, *Software: Practice and Experience*, *2022;00:1–6*. https://doi.org/10.1002/spe.3134