Prompt Application-Transparent Transaction Revalidation in Software Transactional Memory

Simone Economo, Emiliano Silvestri
Pierangelo Di Sanzo, Alessandro Pellegrini
DIAG - Sapienza University of Rome
Rome, Italy
{economo,silvestri,disanzo,pellegrini}@dis.uniroma1.it

Francesco Quaglia
DICII - University of Rome Tor Vergata
Rome, Italy
francesco.quaglia@uniroma2.it

***Abstract*—Software Transactional Memory (STM) allows encapsulating shared-data accesses within transactions, executed with atomicity and isolation guarantees. The assessment of the consistency of a running transaction is performed by the STM layer at specific points of its execution, such as when a read or write access to a shared object occurs, or upon a commit attempt. However, performance and energy efficiency issues may arise when no shared-data read/write operation occurs for a while along a thread running a transaction. In this scenario, the STM layer may not regain control for a considerable amount of time, thus not being able to early detect if such transaction has become inconsistent in the meantime. To tackle this problem we present an STM architecture that, thanks to a lightweight operating system support, is able to perform a fine-grain periodic (hence prompt) revalidation of running transactions. Our proposal targets Linux and x86 systems and has been integrated with the open source TinySTM package. Experimental results with a port of the TPC-C benchmark to STM environments show the effectiveness of our solution.**

I. Introduction

Transactional Memory (TM) allows marking software tasks as *transactions* guaranteed to be executed in isolation and with all-or-nothing guarantees, thanks to the capabilities of some underlying TM layer. The latter can be implemented either via software—this is the case of Software-TM (STM) [6]—or via hardware support, as for the case of processors bundling Hardware-TM (HTM) facilities [1]. TM synchronizes shared-data accesses transparently to the application code, while enabling scalability levels that are close to those achievable via application-specific manual coding with fine-grain locking schemes. On the downside, TM may lead to waste of CPU time because of transaction aborts.

To cope with this latter aspect, in this article we present an innovative software architecture that allows a fine-grain, periodic, and transparent revalidation of running transactions—hence prompt abort of the transaction, if inconsistencies are detected—in scenarios where the application execution profile is arbitrary. This problem is non-trivial to cope with because of the following reasons:

- to actually reduce wasted time, the periodic revalidation mechanism must be lightweight, otherwise its cost will exceed any potential revenue;
- in-memory transactions are typically fine-grain, so prompt revalidation needs to be executed with relatively high frequency, independently of the actual execution profile of the application code.

Devising an application-transparent revalidation support satisfying the above requirements is, to the best of our knowledge, a not yet pursued challenge. In current STM implementations (see, e.g., [2]) transaction revalidation is only performed upon explicit shared-data accesses, if such access may result into a violation of the opacity correctness criterion. However, if a thread runs a transaction that does not access shared-data for a while (e.g., it manipulates local variables into the stack), or accesses data deemed valid, then there is no possibility to quickly react to the materialization of inconsistencies in the running transaction. A possibility is to use mechanisms such as (temporized) operating system signals for bringing control back to the STM layer at arbitrary (e.g., periodic) points in time. However, the delivery of such signals exhibits the granularity of the operating system tick-interval—e.g., 1 to 4 milliseconds in classical Linux configurations—while in-memory transactions can show (much) finer granularity.

To cope with the above issues, we designed and implemented an STM architecture targeting Linux and x86 systems, which is able to exploit very fine-grain hardware-timer events, directly delivered to (and handled by) user-space code. This architecture allows a running thread to change its execution flow and carry out the revalidation task independently of the application code willingness to explicitly pass control to the STM layer. This enables a timer-based revalidation at execution points not covered by the traditional approach. Notably, our solution adds new capabilities, in terms of support for early aborts of transactions, which are orthogonal to those already offered by traditional STM platforms that revalidate transactions only upon read/write accesses to shared-data. To pursue both correctness and performance goals, these two mechanism need to be combined in a synergistic manner, as it occurs in our architecture. We also report the results of an experimental assessment of our proposal, which has been based on running a port of the TPC-C benchmark to STM.

The remainder of this article is structured as follows. In Section II we discuss related work. Our innovative revalidation architecture is presented in Section III. Experimental data are reported in Section IV.

II. Related Work

One major research trend in TM systems is the one of reducing as much as possible the incidence of transaction aborts. In this context, *transaction scheduling* policies are used to control whether some standing transaction can be admitted to the processing stage, or needs to be delayed for a while, because of a high likelihood of conflicts with already running transactions. *Thread scheduling* policies stand as an alternative approach to the reduction of the incidence of aborts. They aim at (dynamically) determining the well-suited thread-level parallelism of TM-based applications. A recent survey of all these techniques can be found in [5].

In [4] the authors propose a solution for enabling a no-longer consistent transaction to be rolled-back partially (rather than totally), which can help saving work otherwise doomed to be unfruitful. In this proposal transaction revalidation is only actuated upon a new explicit access to shared-data by the application code, which is intercepted and handled by the STM layer.

Our work is orthogonal to all the above solutions, since our objective is to enable a thread running a transaction to transparently change its execution flow for revalidation purposes independently of the actual operations carried out within the transactional context.

III. Prompt Transaction Revalidation

A. Baseline Components

1) TinySTM: Our prompt transaction revalidation architecture has been integrated with the open source TinySTM [2] package, although its design principles are general. In fact, it embeds a skeleton that enables a thread currently running a transaction to transparently change its execution flow with fine-grain period, in order to launch an arbitrary user-space callback routine—a revalidation routine in our case. TinySTM manages transactions by relying on a *global version clock* (*gvc*). It is a global shared counter atomically incremented whenever a thread commits a transaction that updates shared-data. A data object is a memory word, and each word address is associated with its own meta-data consisting of (A) a lock-bit and (B) a timestamp, both kept in a single entry of a hash array that is manipulated atomically (also called *lock array*). When a transaction commits, the updated *gvc* value is reflected as the new timestamp of the written word. Upon (re)starting a transaction, a thread stores the current value of the *gvc* into a local variable called *transaction start-timestamp* (*tst*). Upon a write operation, the target address and the value to be stored are both added to the transaction write set. Read operations on shared objects previously updated by the same transaction are served by picking values from the transaction write set. Instead, read operations performed on shared objects outside the write set lead to sample the timestamp and the lock bit of the shared object in order to check if (A) the timestamp is less than or equal to the *tst* of the reading transaction, and (B) the object is not currently locked. If both checks succeed, it means that no concurrent transaction has modified the object in the interval between the start of the reading transaction and the actual read operation, hence the read value is *valid*. Otherwise, the transaction gets aborted.

A mechanism that is used in combination with this scheme is called *snapshot extension*. When the thread reads an object whose timestamp is greater than *tst*, this mechanism checks if all the previously executed transactional read operations (if any) are still valid in a snapshot that includes the timestamp of the culprit read. If yes, the snapshot seen by the transaction is still consistent, hence the transaction is not aborted. Additionally, the *tst* is updated to the *gvc* value sampled immediately before performing the check. In such a case, since the abort is avoided, the transaction can continue its execution.

Clearly, such a revalidation policy does not allow timely revalidations when the transaction does not perform read accesses to shared-data for a while—e.g. it performs other operations such as accesses to private data into the stack—or in the case of a new read that does not trigger a snapshot extension. Our prompt revalidation approach exactly solves this problem, which is relevant when considering that the actions performed by the application code within transactional context (accesses to shared objects vs. accesses to private data) are essentially arbitrary and only related to the way the application logic is implemented.

It must be noted that our prompt revalidation architecture operates synergistically with the aforementioned snapshot extension mechanism. Periodic revalidation can be skipped in our modified STM runtime when *tst* and *gvc* are discovered to have the same value (see next section). As a side effect, our solution can also anticipate a portion of the extensions that would be performed by TinySTM spontaneously, thus making a higher fraction of reads fit into the current transactional snapshot. As such, our revalidation task coincides with the `stm_extend` API offered by TinySTM.

Upon attempting to commit a writing transaction[1], all the reads on shared objects are revalidated and a timestamp corresponding to the value $gvc+1$ is installed for each written object in the respective entry of the lock array. If the revalidation succeeds, and provided that the thread managed to take the locks for each written object[2] the write set is committed to memory.

2) The `dev_extra_tick` *Linux Module:* Another component we exploit in our architecture, although in a re-devised version suited for our purposes, is the `dev_extra_tick` Linux loadable module presented in [3]. It allows to dynamically control the LAPIC-timer on board of x86 processors to enable the original operating system tick assigned to a thread to be partitioned into finer grain extra-ticks, according to a configurable *scale* parameter. This mechanism does not change the planning of CPU usage among threads that is put in place by the Linux kernel. Rather, the only effect of

[1]For read-only transactions the commit operation is unnecessary as no shared object must be updated.

[2]Depending on the chosen locking strategy, this may happen at the time shared-data are transactionally written (ETL, or Encounter-Time Locking) or at commit time (CTL, or Commit-Time Locking).

```c
typedef struct _control_buffer {
  // thread is running in transactional context
  unsigned int  tx_on;

  // revalidation took place since the last extra-tick
  unsigned int  recently_validated;

  // asynchronous revalidation
  unsigned int standing_tick

  // first address of application code
  void      *range_start;
  // last address of application code
  void      *range_end;

  // timestamp of the transaction, if active
  timestamp tst;
} control_buffer;
```

(a) The control buffer data structure

```c
int extra_tick_delivery_check(control_buffer *buff){
    int rv = buff->recently_validated;
    buff->recently_validated = 0; // This is reset anyway

    // The extra-tick must be filtered if any condition
    // in the sequence of checks is satisfied
    if (!buff->tx_on) return 0
    if (!(buff->tst < *gvc)) return 0;
    if (rv) return 0;
    if (!range_ok()) {
        buff->standing_tick = 1;
        return 0;
    }
    // The extra-tick must be delivered through a control flow
    // variation along the running thread
    return 1;
}
```

(b) The extra-tick filtering function

Fig. 1: The extra-tick filtering logic and data structures

an extra-tick delivery is the one of resuming the user-space execution of a thread from an instruction at a chosen address, which corresponds to the address of some *extra-tick handler*. To achieve this, a thread must request extra-tick deliveries to the kernel by explicitly registering itself via the `ioctl()` system call. Benchmarking data provided in [3] show that this module induces negligible overhead for extra-tick deliveries on commodity hardware, even under very short extra-tick periods of the order of 50 microseconds. In the experimental section we report further overhead data when using extra-ticks as exploited in our STM-oriented management mechanism.

B. System Architecture and Implementation Details

A core part of our prompt revalidation architecture has been based on modifying the `dev_extra_tick` module in order to tailor it to the need of STM environments. In particular, we modified the kernel-level handler of the hardware-timer interrupt in such a way that only a subset of the issued extra-ticks are actually delivered to user-space code for performing transaction revalidation. The decision on whether to impose a change of the current execution flow of the thread for revalidation purposes is therefore directly actuated at kernel level in a lightweight manner. Specifically, the extra-tick filtering mechanism avoids bringing control back to the STM layer if:

a) the thread running the application is not currently executing a transaction;
b) the thread is running in a transactional context but the transaction is surely consistent;
c) the thread is running in a transactional context but the transaction has been already (very) recently revalidated;
d) the thread is running in a transactional context but control is currently within an external library (such as the C standard library) or non-application code (such as within the STM layer itself).

As for point d), we note that interrupting the execution of some "environmental" function to carry out revalidation would require that function to be reentrant along the same thread, which is not always guaranteed. The revalidation task could indeed rely on the same function (e.g. a memory allocation function) before the previous invocation has returned. This is true for many standard-library functions, system calls and for functions within the STM layer itself. Filtering the extra-tick delivery, so as to avoid interrupting the execution flow of a thread while running environmental code, allows us to comply with any kind of environmental software. Also, it makes our solution easily reusable independently of the actual environmental services (if any) used by the STM layer itself for carrying out the revalidation task. On the other hand, simply filtering the extra-tick delivery would lead to lose the rhythm of fine-grain revalidation. This issue is tackled in our architecture via a capability we refer to as *asynchronous* revalidation (with respect to the extra-tick arrival), which we shall discuss shortly.

In order to discriminate and manage the scenarios in points a)-d), we have re-engineered the `dev_extra_tick` module so as to enable a thread registered to be extra-ticked to communicate to the kernel the address of a *control buffer* that is used to program and assist the delivery of extra-ticks to user-space code. In particular, the control buffer can be programmed to notify the hardware-timer interrupt handler supported by `dev_extra_tick` whether any of the conditions in previous points are in place. The structure of the control buffer is presented in Figure 1a.

The control buffer content is managed in an application-transparent manner via wrappers around the STM API functions invoked by the application code. In particular, the `stm_start` API offered by TinySTM has been wrapped so that, right before returning control to the application layer, it sets to 1 the flag `tx_on` within the thread control buffer, and updates the `tst` field to the timestamp value assigned by the STM layer to the transaction. Similarly, the `stm_commit` API has been modified to reset the `tx_on` flag.

The `recently_revalidated` flag works like a sticky flag to check whether revalidation has been already executed in between two consecutive extra-ticks. After being initialized to the value 0 by an invocation to `stm_start`, it is set to 1 each time a revalidation not triggered via extra-ticks

takes place, e.g., because of a snapshot extension attempted by the STM layer. This flag is checked by the hardware-timer interrupt handler at kernel level to determine whether the extra-tick needs to be actually delivered to the STM layer, and is reset by the same handler (regardless of any possible filtering condition) as a way to capture a notion of "time proximity" with respect to the last occurred revalidation.

The other two fields in the buffer, namely `range_start` and `range_end`, represent the range in which application code is located in memory—these are the boundaries of the application-specific modules within the text section of the executable, and are easily determined at compile/link time of the application. When a transaction is interrupted in code living outside of such range, the hardware-timer interrupt handler notifies via the control buffer that it could not deliver that extra-tick synchronously, and that such tick may be processed by the STM layer in an asynchronous manner. This is exactly the purpose of the `standing_tick` field. Asynchronous processing takes place in our architecture by wrapping public STM functions in TinySTM. The purpose of these wrappers is to check if there is a standing tick and invoke the revalidation task, if the flag is raised, upon function return. In other words, this scheme leads our architecture to process an extra-tick—which could not be delivered synchronously—as soon as possible, i.e., right before returning control to the application code after an invocation to some STM function[3]. In this way, the rhythm of fine-grain revalidations tends to be preserved independently of the interleaving between application and environmental software along the same thread.

The kernel module is also notified of the memory address where the global version clock *gvc* kept by TinySTM resides, to compare *gvc* against the `tst` field within the thread control buffer[4]. Most specifically, when `tst` is equal to *gvc* no concurrent commit can have occurred, therefore the running transaction is surely consistent and the interrupt handler can simply return.

Overall, the high-level functioning of the filtering logic we embedded within the hardware-timer interrupt handler is encapsulated within the kernel function `extra_tick_delivery_check` and works as in Figure 1b. The concrete kernel module implementation needs to handle additional low-level details to make sure that performance aspects are taken into account. To name a few of such aspects, all data movements from and to the control buffer (which resides in user-space memory) are regulated by the fast `__put_user` and `__get_user` kernel functions, which are deprived of any check for different code segments or insufficient page permissions. Besides, the whole range of addresses that make up the control buffer is properly `mlock`'d so as to avoid having to swap the respective virtual pages in RAM along the critical path of the hardware-timer interrupt handler.

[3]In our architecture we also offer wrappers for the most-used standard-library functions, as a way to reduce the latency of asynchronous extra-tick processing. This allows coping with a wide range of diverse application coding schemes, where the interaction between application code and STM API can be more or less prominent compared to the API offered by the standard library.

[4]In our implementation this field is updated each time a snapshot extension occurs.

C. The Extra-tick User-space Callback

A complementary component of our fine-grain extra-tick delivery is represented by the extra-tick user-space callback. It is the function that is invoked every time an extra-tick successfully passes through all kernel-level filters and is delivered to user-space. Thanks to the hardware-timer interrupt mechanism described in the previous section, the invocation of the callback function is transparent to the application and has the same effects as placing an explicit call to this function in the application code. The important difference is that the kernel-level interrupt handler places such calls dynamically along the application code lifetime, and at periodic intervals. Achieving the same dynamism and degree of precision using alternative techniques (such as with dynamic binary software instrumentation) would likely produce higher overheads at no additional benefit.

A major drawback of this dynamic scheme is that the compiler cannot be aware of these run-time callback activations and therefore does not make sure that the execution state is properly preserved across such dynamic calls. In particular, the current stack frame and value of registers are likely to be exposed to unexpected writes in the extra-tick callback, thus producing unrecoverable damages to the execution state to be resumed upon callback return. To comply with all the above issues, we adopted the following strategy. First of all, stack frames are preserved in our software architecture by having all application code automatically compiled with the `no-red-zone` flag. This flag tells the compiler that all function invocations must have an explicit stack frame and that the address of any stack location in the current frame must be greater than the stack pointer itself. On the other hand, the value of registers is preserved by making sure that the extra-tick callback handler is invoked within a *trampoline sequence* which, regardless of the calling conventions adopted in the application code, saves and restores the values of all user-visible registers, including the status register. Therefore, the extra-tick callback seen by the kernel module is the trampoline itself, which eventually invokes the `stm_extend` API of TinySTM for performing transaction revalidation.

To minimize the cost for saving CPU registers upon the activation of the extra-tick callback, we also exploited low-level optimizations such as saving/restoring the status register via `lahf`/`sahf` instructions, as well as using `fxsave` and `fxrstor` to efficiently copy floating-point registers.

Notice that, in case of asynchronous delivery of standing extra-ticks, the wrappers we developed for public STM functions in TinySTM just call `stm_extend` before they return, with no need to save the processor state since the compiler guarantees that such explicit function calls are correctly managed—in terms of saving/restoring CPU registers—in compliance with the specific reference ABI.

IV. Experimental Evaluation

In this section we report results assessing the effectiveness of our architecture[5]. We run experiments on a cluster of two 64-bit NUMA HP ProLiant servers, each equipped with four 2GHz AMD Opteron 6128 processors and 64 GB of RAM. Each processor has 8 cores, for a total of 32 CPU-cores. The STM application is deployed on one of the nodes—acting as a back-end data layer—while the other node is used for generating the workload of transactional requests. The operating system is OpenSuse 13.2, with Linux kernel 3.16.7. To study the effects of transaction revalidations occurring at different frequencies along the application lifetime we have used three different extra-tick intervals, namely 200, 100, and 50 microseconds. They correspond to $\frac{1}{5}$, $\frac{1}{10}$, and $\frac{1}{20}$ of the original operating system timer-tick of 1 millisecond, or equivalently to a *scale* parameter for the `dev_extra_tick` module of 5, 10, and 20. Plots refer to such experiments with the common prefix of PR (PROMPT-REVALIDATE), followed by the suffix *s5*, *s10*, or *s20* depending on the reference scaling factor.

As a benchmark application, we have used a port of TPC-C [7] to the STM environment. TPC-C is representative of OLTP workloads and includes different transaction profiles that simulate a whole-sale supplying items from a set of warehouses to customers within sales districts. In our experiments we instantiated one district, and generated a workload made up by requests spanning four different transaction profiles specified by the benchmark, excluding the "delivery" profile since it is conceived to be run in deferred mode as per TPC-C specification.

We note that transactions belonging to different profiles exhibit very different CPU demands and different data access patterns, thus enabling a study of our proposal with an articulated workload. In our porting to the target STM environment, CPU demands range from tens of microseconds to milliseconds, as shown in the table below, where we also report the percentage mix of the different profiles.

ID	Profile	CPU demand	% mix
1	**new order**	$\approx$ 350 μsec	0.49
2	**payment**	$<$ 10 μsec	0.43
3	**order status**	$\approx$ 10 μsec	0.04
4	**stock level**	$\approx$ 650 μsec	0.04

Further, this diversity in the granularity of the different profiles allows assessing our prompt revalidation architecture against a non-favourable workload, which also includes transactions that are highly unlikely to be hit by extra-ticks, given their very fine-grain nature (i.e. **payment** and **order status**). Hence, in this scenario, the capability of our architecture in terms of possibility to hit running transactions via extra-ticks is limited to a fraction of the overall workload.

We run our experiments with continuous injection of transactional requests, using either 8, 16 or 24 threads for processing the requests on the back-end data management node, and 6 threads for managing the socket pool from which the client-generated workload comes. This scenario led to use at most

[5]Available at https://github.com/HPDCS/tinySTM-reval

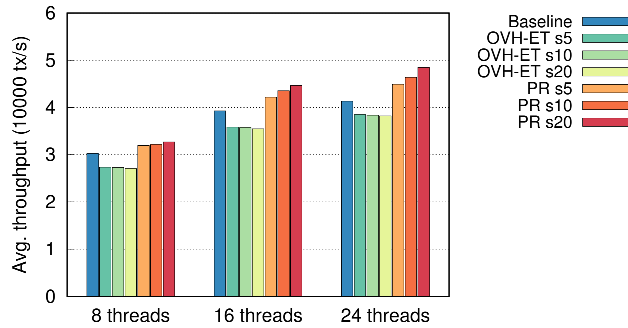

Fig. 2: Throughput data.

94% of the CPU computational power at the server side, thus avoiding hardware resources saturation that would affect the reliability of the experimental analysis. We decided to vary the number of threads used to process transactions in order to assess our proposal with different levels of actual transaction concurrency. In any case, at each thread count we always run with the highest concurrency since we configured TinySTM to rely on the CTL scheme (rather than ETL) for data-lock acquisition. In all the configurations, we set the backlog of pending transactional requests to be processed at the server side to 4096, and we experimented with a sustained workload leading the backlog to be close to saturation at any used thread count. Each experiment with 8 threads entails 1 million committed transactions, while all the experiments with 16 and 24 threads entail 2 and 3 million committed transactions, respectively.

In Figure 2 we report the transaction throughput that we observed in the different configurations, plus a baseline experiment where our prompt revalidation architecture is disabled. Each histogram refers to an average over 10 runs of the same configuration (variance is not reported since the results for different runs where within the 2% of each other, except for profile 3). We have also included the throughput observed when running the system without installing the extra-tick user-space callback (denoted OVH-ET in the plots), as a way to assess the overhead introduced by our kernel module (particularly, by the extra-tick management mechanism).

By the results we see that the prompt revalidation architecture allows improving the system throughput, compared to the baseline, by up to 17% and up to roughly 8000 additional txs/sec in absolute terms. The maximum gain is noted for the 24 threads case, meaning that our prompt revalidation mechanism leads to better exploit the increased parallelism in the execution of transactions. As for OVH-ET, it can be seen that it shows no more than 9% worse performance than the baseline. More important, such performance loss tends to slightly scale down at larger thread counts. Overall, the data suggest that our mechanism provides better benefits in the relevant scenario where there is a high degree of actual transaction parallelism, which gives rise to many conflicts that cannot be detected in time by the underlying STM platform in case of the baseline revalidation scheme. Moreover, the

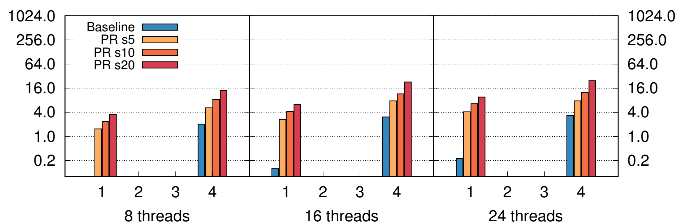

Fig. 3: Number of successful validations with snapshot extension per commit (y-axis) per transaction profile (x-axis)

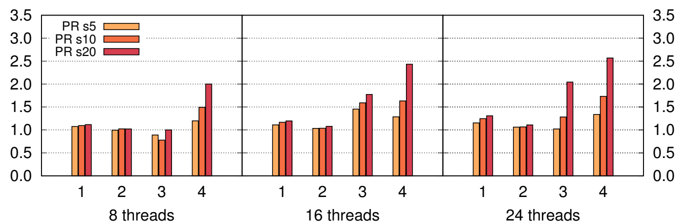

Fig. 4: Total number of aborts (y-axis) per transaction profile (x-axis) relative to baseline

overhead results show that we are able to consistently defeat the actual overhead of the hardware-timer interrupt and of its management logic, and that it does not undermine the benefits of our prompt revalidation mechanism.

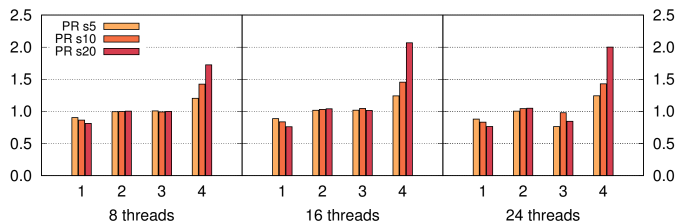

Fig. 5: Average turnaround (y-axis) per transaction profile (x-axis) relative to baseline

A second batch of experimental results are reported in order to show the performance of our system in terms of: 1) increase in the number of validations with snapshot extension per commit, 2) variation in the number of aborts per profile, and 3) improved turnaround per profile. The number of validations with snapshot extension per commit (Figure 3) is definitely increased compared to the baseline, with values that grow steadily while moving from *s5* to *s20*. This illustrates that our architecture is much more capable to check the consistency of ongoing transactional work and re-evaluate running transactions. As for the aborts (Figure 4), we can see that our prompt revalidation scheme gives rise to no more than 130% of the aborts experienced by baseline for the two transactional profiles that constitute the major portion of the overall workload, namely profiles 1 and 2. However, a higher number of aborts does not necessarily imply a longer turnaround time for completing a transaction. In fact, in Figure 5 we see that the average turnaround per profile[6] is reduced by up to 25% for the most relevant profile—namely, the **new order** transaction, which is long running and has a very relevant weight in the workload mix. Also, a significant worsening of the turnaround is noted only for the **stock level** transaction, which has a marginal weight it the workload mix. Overall, turnaround data are consistent with throughput data.

V. Conclusions

In this article we have presented and evaluated a mechanism for early detecting conflicts in Software Transactional Memory (STM) applications, which is based on the application-transparent prompt revalidation of running transactions. Such mechanism relies on a software stack we implemented that delivers fine-grain hardware-timer interrupts directly to user-space code—more precisely, the STM layer—in order for it to gain control independently of the activities currently carried out in a thread. If that thread is running a transaction that is already doomed to abort, its inconsistency is promptly detected by the STM layer, thus leading to an early abort that reduces the CPU cycles spent for inconsistent work. To achieve a better trade-off between effectiveness and overhead of our software stack, we have devised a filtering mechanism that prevents the delivery of hardware-timer interrupts to the STM layer when either the transaction is still surely consistent, or a revalidation has been already performed recently. Experiments carried out with a port of the TPC-C benchmark to STM suggest that our solution is effective in reducing the waste of CPU cycles for inconsistent work, thus improving the delivered performance.

References

[1] http://www.intel.com/content/www/us/en/processors/core/5th-gen-core-processor-family.html.

[2] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, pages 237–246, 2008.

[3] A. Pellegrini and F. Quaglia. A fine-grain time-sharing Time Warp system. *ACM Transactions on Modeling and Computer Simulation*, 27(1): 10:1-10:25, 2017.

[4] A. Porfirio, A. Pellegrini, P. di Sanzo, and F. Quaglia. Transparent support for partial rollback in software transactional memories. In *Proceedings of the 19th International Conference on Parallel Processing*, pages 583–594, 2013.

[5] P. D. Sanzo. Analysis, classification and comparison of scheduling techniques for software transactional memories. *IEEE Transactions on Parallel and Distributed Systems, 10.1109/TPDS.2017.2740285.*

[6] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[7] TPC Council. TPC-C Benchmark, Revision 5.11. Feb. 2010.

[6]Turnaround is the latency from the start of transaction processing up to its commitment, thus including the latency for intermediate aborted runs.