Preemptive Software Transactional Memory

Emiliano Silvestri, Simone Economo, Pierangelo Di Sanzo, Alessandro Pellegrini, Francesco Quaglia
DIAG–Sapienza Università di Roma, Italy

***Abstract*—In state-of-the-art Software Transactional Memory (STM) systems, threads carry out the execution of transactions as non-interruptible tasks. Hence, a thread can react to the injection of a higher priority transactional task and take care of its processing only at the end of the currently executed transaction. In this article we pursue a paradigm shift where the execution of an in-memory transaction is carried out as a preemptable task, so that a thread can start processing a higher priority transactional task before finalizing its current transaction. We achieve this goal in an application-transparent manner, by only relying on Operating System facilities we include in our preemptive STM architecture. With our approach we are able to re-evaluate CPU assignment across transactions along a same thread every few tens of microseconds. This is mandatory for an effective priority-aware architecture given the typically finer-grain nature of in-memory transactions compared to their counterpart in database systems. We integrated our preemptive STM architecture with the TinySTM package, and released it as open source. We also provide the results of an experimental assessment of our proposal based on running a port of the TPC-C benchmark to the STM environment.**

I. Introduction

Transactional Memory (TM) is the raising paradigm for the management of shared-data accesses on multi-core machines. It allows programmers to mark code blocks as transactions, which are then handled, in terms of actual memory operations, by some underlying TM layer. The latter is in charge of guaranteeing isolation and atomicity while executing those code blocks, i.e., all or nothing execution semantic. This allows achieving similar or better level of performance than fine-grain hand-made locking. Anyhow, TM jointly guarantees much higher transparency to the programmer since she is fully relieved from the burden of hand-coding the synchronization logic.

Nowadays various TM implementations exist, including the ones natively embedded in modern processors via specific hardware support, the so-called Hardware-TM (HTM) [1]. However, the most diffused implementations are still based on software support, known as Software-TM (STM) [18]. These provide the advantages of not requiring any specific hardware technology. Further they bypass several important limitations of current HTM implementations, such as the impossibility to commit a transaction that undergoes a user/kernel mode switch or whose write set exceeds the size of the L1 cache[1]. However, despite the offered advantages, STM environments are still doomed to improvements, particularly for what concerns the management of differentiated transaction priority levels.

The difficulty in handling multiple priority levels in STM systems comes out since it is generally not convenient to run TM applications with a number of threads exceeding the number of available CPU-cores, mostly because they show a CPU-bound execution profile. In fact, in-memory transactions make scarce usage of blocking Operating System services, like I/O system calls. Therefore, the dynamic spawning of a new thread as a reaction to the arrival of some high priority request to run an in-memory transaction might be inviable, both because of the overhead for the spawn operation and because it would lead to the scenario where multiple threads compete for CPU-usage, which has been shown to be likely adverse to TM applications [10] ([2]). On the other hand, resorting to a static pool of threads for processing higher priority requests, each one bound to a given CPU-core, might give rise to CPU under-utilization along execution phases not showing the presence of higher priority requests. Consequently, in their common implementations, TM systems simply delay the processing of an incoming high priority request up to the point in time where some thread ends its last started transaction and runs the routine in charge of verifying the presence of the new request.

We note that, even if a signaling mechanism were used to notify the materialization of a new request, such as Posix user-defined signals, the timeliness of the signal delivery to the destination thread would be still bound to the conventional Operating System timer-interrupt interval, thus resulting not fully adequate for promptly dispatching high priority transactional requests. This aspect is also linked to the finer-grain nature of TM transactions (compared to their counterpart in database systems), which is originated, among other, from the absence of I/O interactions along the

[1] Off-the-shelf HTM implementations keep the write set of the transaction temporarily buffered at the cache level, and flush it to lower levels in the memory hierarchy only upon a successful commit. On the other hand, transactional updates are squashed if an interrupt is received by the CPU-core running the transactional code block, which gives rise to the abort of the transaction.

[2] Assigning higher CPU scheduling priority at the Operating System level to threads running higher priority transactions would be a means to alleviate such a problem, but not a definitive solution for managing thread competition for CPU usage. Also, it might lead to starvation of lower priority transactions uncontrollable by the STM layer.

in-memory transaction lifetime. In more detail, a conventional timer-interrupt interval (typically ranging from 1 to 4 milliseconds on most Operating System configurations, such as Linux ones) would still delay too much the activation of the signal handler able to detect the presence of the standing high priority transactional request.

In this article we cope with the above problem by presenting the design and implementation of a preemptive STM environment to be run on top of Linux/x86 systems. In our software architecture we exploit an ad-hoc timer management Linux module, originally presented in [14], which allows for fine-grain periodical control flow variations along any running thread with no intervention by the chain of kernel-level mechanisms used for supporting Posix signals, hence leading to minimal run-time overhead. In our architecture, this module is exploited to make a thread that has already dispatched a low priority in-memory transaction promptly switch to the execution of some standing higher priority one.

Our solution is also based on a fully new management scheme of differentiated execution contexts within the STM layer so that the transaction context-switched off the CPU is not aborted. Rather it will be eventually resumed so that its outcome will be only determined by possible data conflicts, as typical of the STM paradigm. Overall, in our approach we promptly nest the execution of the higher priority transaction along an already active thread, with no need for additional threads, thus preventing at all the aforementioned problems related to CPU competition by multiple threads in TM systems. Additionally, we avoid CPU under-utilization that would be caused by statically destining specific threads to process higher priority requests, given that in our architecture each threads can be in charge of processing whichever in-memory transaction at any time instant (either a new standing one with higher priority or a previously context-switched one).

Great attention in our design is put on the data structures and logic for carrying out the preemption mechanism. This enables us to provide a preemptive STM architecture with minimal overhead at the side of both kernel (via the aforementioned lightweight timer-interrupt handling) and user space software.

Our proposal does not create any bias in terms of CPU assignment across threads (including kernel-level threads) running on top of the Linux system. In fact, the fine-grain timer-interrupt mechanism we adopt for threads running the STM application does not alter the original Operating System planning in terms of overall CPU time to be assigned to the different threads. It only allows an original tick destined to those threads to be partitioned into subintervals, at whose end a control flow variation leading control to a higher priority transaction, if any, may occur. This prevents impairing fairness when running our preemptive STM system on top of a multi-user conventional platform.

We have integrated our implementation with the open source TinySTM package [11] and released it as open source as well. In this article we also report the results of an experimental assessment of our proposal based on 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. The preemptive STM architecture is presented in Section III. Experimental data are provided 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. Along this path, several approaches have been based on so called *transaction scheduling* policies [3], [7], [22], which 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. A few of these techniques [8] rely on migrating transactions to queues managed by different threads, so as to increase the likelihood that transactions accessing overlapping data sets are serialized, thus not interfering with each other. An alternative approach to the reduction of the incidence of rollback has been the one of adopting *thread scheduling* policies [6], [9], [15]–[17]. Unlike transaction scheduling, thread scheduling policies do not delay the processing of standing transactions. Rather, they aim at (dynamically) determining the well-suited level of parallelism, which is the one that avoids thrashing due to excessive transaction aborts caused by oversized thread-level concurrency. Other techniques have been oriented to the optimization of the strategy for managing contention across concurrent transactions [5], [21]. Some of these approaches also enable the run-time adaptation of the contention strategy to the workload profile [5]. The orthogonal issue of mapping threads to CPU-cores in TM applications for performance optimization has been addressed in [2].

Our work is orthogonal to all these approaches since none of them copes with transaction priorities, and with the possibility to timely pass control to higher priority transactions via interrupt mechanisms.

The only solution we are aware of which discriminates between transaction priorities in TM systems is the one in [13]. In this work the authors cope with quality of service in STM applications and propose an approach where transactions that are subject to deadlines, and experience abort retries due to conflicts, tend to execute more conservatively (e.g., by eager locking data) while getting closer to their deadlines. Implicitly this proposal enables the dynamic increase of the priority of transactions running closer to their deadlines, since eager locks will lead these transactions not to be aborted because of data conflicts. In any case, this work does not make systematic use of preemption in order to enable the timely processing of higher priority transactions along

the threads running the TM application, which is instead the fulcrum of our proposal.

User Level Threads (ULT) [19] is the historical technology enabling time interleaved execution of different code blocks along a same thread, just like we do in our preemptive STM architecture. However, ULT is not application transparent since the programmer needs to inject calls to ULT API functions at specific points of the application code. Our approach is instead fully transparent, so that the programmer of the STM application does not need to care about the management of transaction priorities and control flow variations. She only needs to code the data access logic, while the actual passage of control to higher priority transactional requests is achieved in our architecture via actions performed by the run-time environment. Also, given that preemption of lower priority transactions takes place on the basis of fine-grain hardware timer-interrupts, we also avoid at all context switch delays that would be potentially experienced in some hypothetical architecture based on ULT in scenarios where the lower priority transaction currently running along a thread does not timely reach the point of the call to the ULT API functions.

Finally, we remark again that our work copes with the need for managing differentiated transaction priority levels in a scenario where dynamically spawning (or resuming) higher priority threads for processing these transactions is not viable. This is because of both the overhead/latency for the spawn (or resume) operation and CPU competition that adversely affects performance when running TM systems with more threads than CPU-cores [10]. Also, as already stated, reserving CPU-cores for running threads bound to higher priority requests does not pay off in execution phases where no high priority request is issued. With our approach we can run with a number of threads not exceeding the available CPU-cores, being still able to timely pass control to standing higher priority requests along these threads, thus avoiding at all the above mentioned problems.

III. The Preeemptive STM Architecture

A. Overview

In Figure 1 we show a high level schematization of our preemptive STM architecture, which is targeted at back-end STM environments. A classical socket pool is handled in order to receive requests for executing data manipulations transactionally, which come in from some front-end system. Upon its receipt, a request is placed into a priority queue, by associating it with the corresponding priority level. With no loss of generality we assume the priority level is explicitly marked within the transactional request, together with the function to be run by the STM environment for serving the request, and its input parameters.

Given that a dispatched transactional request could be preempted and paused in favor of a higher priority request to be timely processed along a same thread, we need to manage

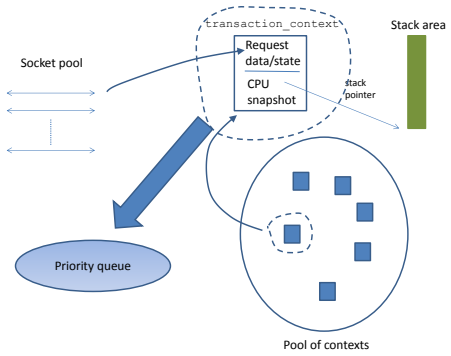

Figure 1. Basic architectural organization.

an individual CPU/stack context for each transaction. A default initial context needs therefore to be associated with each incoming transactional request that is inserted into the priority queue. To set up a pool of contexts to be used for this association, we exploit Posix user-defined signals, with the `SA_ONSTACK` option for the handler. In more detail, we issue `NUM_CONTEXTS` signal instances at startup of the STM environment in order to activate the handler the same number of times. At each activation, the handler makes a snapshot the CPU/stack context into a proper data structure, which we name `transaction_context`. All the set up instances of this data structure go into the pool of contexts to be associated with incoming transactional requests.

When associating a transaction context to an incoming request, the `transaction_context` data structure is also used to keep track of the priority information for the transaction, the function to be run and its parameters, as well as information on what happened along the transaction lifetime (such as the number of times it has been preempted and context-switched off the CPU in favor of a higher priority transactional request). We exploit this information in order to dynamically change the actual priority of a request according to a feedback scheme aimed at improving performance. When a transaction ends its processing phase, the context it is using is released to the pool in order for a subsequent incoming transactional request to reuse the same stack area, in a fresh incarnation of its content.

The value of `NUM_CONTEXTS` is a configurable parameter and determines the maximum number of transactions that are admitted to the processing stage. When no context is available from the pool, incoming transactional requests are not migrated to the priority queue. This migration is resumed as soon as the termination of already active transactions will lead to releasing contexts to the pool. Our architecture is intentionally devised in order to manage more contexts than worker threads, since transactions can be preempted (hence paused) and then resumed. Therefore `NUM_CONTEXTS` should be set to a value significantly greater than the number of worker threads selected for running the STM application.

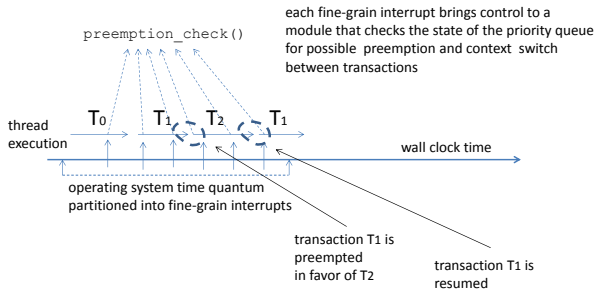

Figure 2. Fine-grain interrupts timeline.

The job of receiving requests from sockets and inserting them into the priority queue is done via dedicated threads, whose execution profile is clearly I/O bound. Hence, request insertion into the priority queue takes place off the critical path of the worker threads running the STM based application logic. This enables to find the most up-to-date state of the priority queue every time a fine-grain periodical control flow variation occurs along any worker thread to verify the need to pass control to some standing higher priority request.

As we show in Figure 2, such periodical control flow variation is based on fine-grain timer-interrupts—with period of the order of tens of microseconds. These are supported at low cost by the ad-hoc Linux module presented in [14] which we exploit as baseline component in our architecture, whose relevant details are provided in Section III-B.

Fine-grain timer-interrupts are issued exclusively towards worker threads and lead to the activation of a user space function—called `preemption_check()`—which implements the preemption management policies at the core of our STM environment. If `preemption_check()` determines that a different transaction needs to take control of the CPU-core, the currently processed transaction is preempted, and its context is enqueued again within the priority queue, while the context of the higher priority transactional request is installed so that the worker thread can start processing it. As soon as a worker thread ends the processing phase of its current transaction, it releases the no-more-in-use context to the pool, and then queries the queueing data structure in order to take care of activating, or resuming in case of a previous preemption, the transaction that currently stands at the highest level of priority, if any. This is what happens to T_1 in the example in Figure 2, which is preempted in favor of T_2 along a worker thread, and then resumed (in this case along the same thread) after T_2 ends.

B. Kernel Support for Fine-grain Timer-interrupts

x86 processors are equipped with a per CPU-core programmable timer device known as LAPIC-timer. Linux configures the LAPIC-timer to generate periodic interrupts according to the frequency established by the `CONFIG_HZ` parameter defined at kernel compile time. Classical interrupt periods range from 1 to 4 milliseconds. To achieve the possibility to deliver finer-grain timer-interrupts to running threads, the Linux module presented in [14] offers the support for a special device file called `dev_extra_tick`. A worker thread can register itself as one to be hit by fine-grain timer-interrupts—also referred to as *extra-ticks*—issuing an `ioctl` call towards the device file.

The portions of the whole Linux kernel which need to know whether a thread is registered and needs to be hit by extra-ticks are: (i) the kernel scheduler, and (ii) the top-half (the very early handling logic) of the timer-interrupt. The Linux module implementing the driver of the `dev_extra_tick` device file is also in charge of redefining the actual behavior of these kernel components, which is achieved via dynamic patching. More in detail, an execution flow variation is injected into the kernel `schedule()` function such that control goes to a `schedule_hook()` routine offered by the external module right before `schedule()` would execute its finalization part (e.g., stack realignment and return). The `schedule_hook()` function will simply execute the same return actions originally planned by the kernel `schedule()` function. However, patching the original scheduler in this way allows the hook to take control when the decision about what thread needs to take control of the CPU-core[3] is already finalized. As a consequence, the hook is able to check whether the thread is a registered one and needs to be extra-ticked. In the positive case, `schedule_hook()` executes the following additional steps:

A) It changes the LAPIC-timer period by scaling it on the basis of a configuration parameter. The scaling factor is what determines the length of the extra-tick interval.
B) It records in a per CPU-core entry of a control table that the current CPU-core is working in extra-tick mode.
C) It records in a per registered-thread entry of a control table a counter of extra-ticks not yet consumed by such a thread within the current time quantum assigned by the Operating System.

The information recorded in step B is exploited for reverting the LAPIC-timer configuration to the original one. This happens when the scheduler passes control to a thread that is not registered into `dev_extra_tick`, while the last running thread was a registered one. In this case, the control record associated with the CPU-core is reset in order to reflect that the CPU-core is no longer operating in extra-tick mode. Note that this approach works also in scenarios where the thread registered within the `dev_extra_tick` device file looses control of the CPU-core because of a passage into a sleep state (e.g., for an I/O interaction). Overall, the above scheme allows restoring the LAPIC-timer

[3]It has actually already taken control of the CPU-core, since we are returning from the scheduling process.

configuration to the original one each time a non-registered thread is (re)scheduled, and independently of any state-transition of registered (hence extra-ticked) threads in the Operating System state diagram.

As for the dynamic patching of the LAPIC-timer interrupt management logic, the Linux module locates the launcher code block of the top-half handler in the kernel memory image and replaces the call to the original top-half with one to a top-half hook function. This top-half hook is in charge of executing the same identical basic actions as those executed by the original top-half procedure—such as acknowledging the accepted interrupt. However, it discriminates if the interrupted thread is a `dev_extra_tick` registered one (i.e., if the thread is subject to extra-tick management) and in the positive case it executes the following actions:

(i) It decreases the extra-tick counter associated with the thread.

(ii) If the counter reaches the value zero, then it means that a whole time quantum has expired. In this case, the top-half hook calls the actual kernel function used to update kernel-level timing information. This mimics the behavior of the original top-half execution path, given that it would trigger the timing information update function exactly at the end of each planned time quantum.

(iii) The top-half hook changes the Instruction Pointer (IP) kept by the processor image registered into the system stack upon interrupt acceptance, so that the interrupted thread will gain control in a proper machine code block upon the restore of that image onto the CPU-core—namely, when returning from the LAPIC-timer interrupt. Consequently, the top-half hook also changes the application-level stack layout of the thread by adding a program-counter return value that will allow that code block to exactly return control to the instruction interrupted by the extra-tick—namely, the original IP value logged into the CPU-context snapshot on the system stack. This is done by exploiting the Stack Pointer (SP) value from the logged CPU-context, which then is also modified in order to reflect the insertion of a new element at the top of the user-level stack.

(iv) Finally, if the extra-tick counter of the thread registered within the `dev_extra_tick` device file reached the value zero—see point (ii)—the thread is again filled with the number of extra-ticks (say N) it is allowed to receive in the next time quantum.

In our preemptive STM environment, the address of the code block that will take control thanks to the instruction pointer variation in point (iii) represents the aforementioned `preemption_check()` function implementing the logic for managing transaction priorities and triggering preemption and context switches (see Section III-C). This address is posted to the kernel when calling the same `ioctl` system call that is used for registering the thread in the `dev_extra_tick` device file as one to be extra-ticked.

C. Data Structures and Policies for Priority Management

The priority queue we use in our preemptive STM environment includes a couple of lists $\langle active, standing \rangle$ for each of the managed priority levels. The $standing$ list keeps all the contexts associated with transactions having a given priority, whose execution has not yet been started—these transactional requests have been delivered but have not yet been admitted to the processing stage along any worker thread. Conversely, the $active$ list keeps track of all the contexts associated with transactional requests at that priority level, which have already been started by some worker threads, and have then been context-switched off the CPU (i.e., they have been preempted).

Within the same priority level, the CPU assignment favors transactions within the $active$ list, so that elements within the $standing$ list, if any, are considered for CPU-dispatch only if the $active$ list is currently empty. The policy for managing each of the two lists is First-In-First-Out (FIFO), so that the oldest transaction in the list is always selected for CPU-dispatch before the others. Overall, the priority level is logically seen as the concatenation of the two corresponding lists, which are managed according to a CPU assignment scheme where a transactional request $T \in active$ is seen as preceding any transactional request $T' \in standing$.

Figure 3 provides a graphical representation of such an organization, where the requests kept by the $active$ list are those to be considered *hot*. In fact they are resumed for processing by the worker threads prior to considering any other request kept by the $standing$ list—namely, the *cold* requests.

A compact bitmap is used to determine whether any given priority level has at least one element within the corresponding $\langle active, standing \rangle$ lists. Hence, as soon as one worker thread accesses the priority queue for determining what is the highest priority level that currently keeps some request to be started (a cold one) or resumed (a hot one), such determination takes place via fast bit-wise instructions.

The separation between hot and cold requests within a given priority level, with hot requests favored over cold ones, has been exploited precisely to keep into account the peculiarities of in-memory transactions handled by common STM layers. More in detail, after the start of a transaction, the longer the length of the time interval for reaching the commit phase, the higher the likelihood of observing a conflict with some concurrent transaction. Specifically, delaying the finalization of an already started transaction—because of a context switch off the CPU—leads to a stretch of the so-called transaction *vulnerability window* [13]. This, in turn, may lead to an increased likelihood of abort, a phenomenon adverse to performance. In the end, keeping the already started transactions as hot records within the $active$

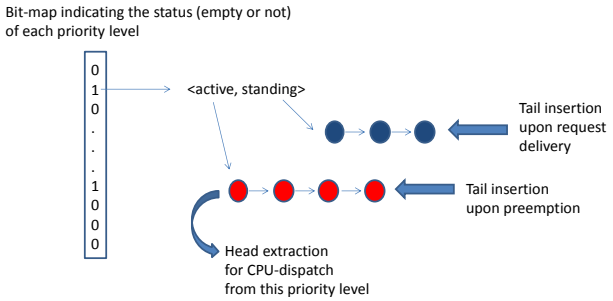

Figure 3. The priority queue.

list, and favoring them over the cold transactions kept by the *standing* list, contrasts the stretch of the vulnerability window. Recall that any transaction within the *standing* list has not yet been started by any worker thread, so that delaying its activation in favor of hot ones has no effect in terms of stretch of its vulnerability window.

On the other hand, the stretch of the vulnerability window of an already started transaction can also be caused by repeated context switches off the CPU caused by the presence of higher priority requests within the priority queue upon running the `preemption_check()` module along the thread. To cope with such an orthogonal problem, we have devised a feedback mechanism such that the actual priority level of an already started transaction is dynamically modified at run-time. In particular, for each already started transaction we keep track of the number of times it has been context-switched off the CPU (preempted) in favor of a higher priority transaction. We denote the counter of context switches off the CPU involving transaction T as C_T. As soon as the value of C_T reaches a threshold that we denote as C_{max}, the transaction is migrated to the highest priority level, so that no further delays caused by preemptions will be induced on it. The responsiveness of such a feedback mechanism depends on the value of C_{max}, since greater values of this parameter will tend not to promote the priority of the transaction along its lifetime. As an extreme, $C_{max} \rightarrow \infty$ leads transaction T to always reside at its original priority level, independently of the number of incurred preemptions. Conversely, setting C_{max} to the minimum value 1 would lead any transaction to reach the maximum priority level right after its first preemption. This, in turn, would lead to flatten the actual priorities of already started (say hot) transactions to the same value, with consequent scarce possibility to discriminate what transaction should actually be processed before the others according to the original priority the transactional requests had upon their delivery.

To keep dynamic priorities more aligned to the original transaction priorities along time, larger values of C_{max} should be selected. On the other hand, we also devised and implemented a variant of the aforementioned dynamic priority assignment mechanism where at each increment of C_T leading the value of this counter to still comply with the inequality $C_T < C_{max}$, we promote anyhow the priority of transaction T by one level. This *lazy priority promoting* scheme has the potential to tackle the stretch of the vulnerability window of an already started transaction, while still not favoring the flattening of the dynamic priorities of active transactions to the maximum priority level admitted in the system. Indicating with P_T the current priority of transaction T, which initially corresponds to the priority level originally assigned to the transactional request, the variation of the priority P_T upon preempting transaction T, with consequent increment of the counter C_T, takes place according to the following scheme:

$$P_T = \begin{cases} min(P_T + 1, P_{max}) & \text{if } C_T < C_{max} \\ P_{max} & \text{otherwise} \end{cases} \quad (1)$$

where we denote with P_{max} the maximum admitted priority level within the priority management scheme.

D. Safe Execution within the STM Layer

One important final aspect to consider relates to how the extra-ticks delivered to threads need to be handled in case they are received while the target thread is currently executing some functions offered by the STM environment or the standard library, rather than native application code. This might be the case when the thread runs the `commit` statement for the transaction it is currently processing, as well as classical `TM_read` and `TM_write` services, which map read/write operations on shared data by the application code to transactional (all or nothing) operations.

Given that these functions might execute critical actions, such as locking data (for instance, several STM implementations rely on the commit-time-locking algorithm [4] which locks, e.g., data in the transaction write set for atomically installing all the newer versions upon a successful finalization), preempting the transaction execution while one of these functions is in progress may hamper both performance and correctness. In other words, we need to leave these functions execute as non-preemptable tasks.

In order to achieve this objective, we have adopted the following strategy. Each worker thread keeps a `PREEMPTABLE` flag on Thread Local Storage (TLS), indicating the state of execution of the thread itself. The flag is set to false each time one of the above functions is invoked by the applications code and is reset to the value true upon returning from the function. This is achieved transparently in our implementation via the reliance on wrappers that are interposed between the application and the STM/standard-library at compile/link time. If the extra-tick is delivered to a thread when the flag is set to false, then the `preemption_check()` function whose activation is triggered by the `dev_extra_tick` device file logic simply

returns. This allows running all the aforementioned functions without any risk of preempting them.

The drawback of this approach is that the delivery of an extra-tick to the worker thread is somehow lost, in terms of its potential for promptly passing control to some higher priority transaction. To cope with this aspect, we added a second per-thread flag, still kept on TLS, named `STANDING_TICK`, which is set to true by `preemption_check()` exactly when an extra-tick is delivered to a thread having `PREEMPTABLE` set to false. `STANDING_TICK` is checked by the wrapper of any non-preemptable function right upon the function return. If it is found to be set to true, then the wrapper resets it and invokes the `preemption_check()` function, which this time will actually run the preemption policy we presented in Section III-C. In other words, if needed we shift the management of preemptions (ideally triggered periodically by the extra-ticks) along the time axis at the earliest point in time such that no critical action is still in place along the worker thread.

A schematization of this behavior is provided in Figure 4. The arrival of the extra-tick at wall-clock-time t_1 triggers the execution of `preemption_check()`. However, given that the `PREEMPTABLE` flag is found set to false because the worker thread previously entered the execution of `TM_write()`, `preemption_check()` simply sets `STANDING_TICK` to true and then returns. Later, upon returning from `TM_write()`, the wrapper resets the flag and calls `preemption_check()` for actual checks on the need for preempting the current transaction.

A minor variation has been put in place to comply with external libraries (e.g., `lib.SO.xx` libraries) that may rely on the usage of the stack *red-zone*, which could not be allowed to be recompiled with the no-red-zone directive upon the installation of our preemptive STM environment[4]. In such a case, the variation of the execution flow of worker threads via the modification of the user space stack of the thread just above the current stack pointer address—in order to activate `preemption_check()`—might damage the stack content. Given that when the extra-tick is received the timer-interrupt hook knows the address of the instruction to be processed upon resuming user space execution, if this address falls outside the memory boundary associated with code portions that do not make use of the red-zone we do not activate `preemption_check()`. Rather, we raise the `STANDING_TICK` directly by the kernel level code embedded within the hook.

[4]The red-zone is the stack region above the current stack frame. It is typically exploited by conventional compilation tool-chains so as to allow a leaf function to use the stack without explicitly reserving storage for the current stack frame.

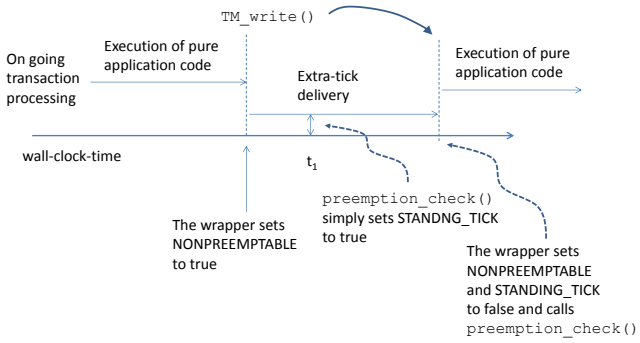

Figure 4. Standing ticks and time shift of preemptions.

IV. Experimental Study

A. Experimental Settings

We run our preemptive STM environment on top of a 64-bit NUMA HP ProLiant server, 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, which share a 12MB L3 cache (6 MB per each 4-cores set), and each CPU-core has a 512KB private L2 cache. The Operating System is OpenSuse 13.2, with Linux kernel 3.16.7.

As stated before, our STM environment has been implemented by using TinySTM [11] as the baseline TM layer and the whole package we developed is available for free download[5]. We note that TinySTM has the possibility to be configured with either encounter-time-locking or commit-time-locking of the data accessed by a transaction. Since we have introduced within TinySTM a fully innovative preemption facility, we decided to experiment with the commit-time-locking configuration, since encounter-time-locking would require the preemptive approach to be complemented with a suitable scheme for managing and resolving priority inversions. This topic is somehow aside of the main contribution provided by our preemptive STM approach, and we plan to study the relation of the two as future work.

In our experiments, we used 16 worker threads in charge of processing transactions, and 5 threads in charge of managing I/O operations on the socket pool and inserting incoming transactional requests into the priority queue. This scenario leads the STM environment to use no more than 65% of the overall available CPU capacity. This choice allows leaving CPU resources for the Operating System—e.g., for kernel level threads in charge of carrying out classical housekeeping operations, such as Linux `kswapd` demons. Hence it allows assessing our proposal in scenarios avoiding interference on the measurements of performance parameters, which could be caused by CPU competition depending on the choices

[5]`https://github.com/HPDCS/PRESTO`

transaction profile	CPU demand	priority level (the higher the better)
delivery	$\approx$ 5 $msec$	1
stock level	$\approx$ 650 μsec	2
new order	$\approx$ 350 μsec	3
order status	$\approx$ 10 μsec	4
payment	$<$ 10 μsec	5

Table I
TRANSACTION PROFILES AND ASSOCIATED PRIORITY LEVELS.

by the Operating System scheduler. The workload generator issuing transactional requests has been run on another multi-core machine with the same technical specifications of the one hosting the STM environment, which we described above. The two machines are connected via a switched 100Mb ethernet.

Finally, the extra-tick interval in our preemptive STM system has been configured to 100 microseconds, a value definitely lower than the timer-interrupt period originally adopted in the configuration of the Linux kernel we used, which was set to 1 millisec. This choice tends to avoid excessive interference by the extra-tick management logic on the operations of the STM system, while still guaranteing that a lower priority transaction will not monopolize a CPU-core while a higher priority one is standing. In fact in this scenario the lower priority transaction will not be allowed to use the CPU-core for more than the very reduced wall-clock-time of 100 microseconds. In any case, results related to the overhead by the extra-tick management logic under these settings are reported in the next section. The size of the pool of contexts has been set to 1024, a value that enables keeping active a number of transactions definitely larger than the number of worker threads processing them.

B. Performance Data with the TPC-C Benchmark

To test our proposal we used a port of the TPC-C benchmark [20] to STM. TPC-C is representative of OLTP workloads and includes 5 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 equally spanning the whole set of the 5 different transaction profiles specified by the benchmark.

It must be noted that transactions belonging to the different profiles exhibit very different CPU demands. In our port to the target STM environment, CPU demands range from tens of microseconds to milliseconds. This peculiarity has been exploited in our experiments in order to determine a transaction priority scheme where shorter-running transactions are given higher priority. We recall that shortest-job-first, with preemption in our case, is a classical way of managing priorities in computer systems, which typically allows the optimization of server side run-time dynamics. As an example, it has been exploited in [12] in order to give higher server-side priority to the transfer of shorter static HTML files in the context of Web-server operations—this is achieved via proper scheduling of socket level operations within the Operating System kernel. Overall, in Table I we report the list of transactional profiles we have exploited from TPC-C in association with the order of magnitude of the CPU demand for processing them and the corresponding priority level we assigned while testing our preemptive STM environment.

We setup the workload generator to inject 25000 transactional requests per second, issuing a total number of 6 millions of transactional requests along the experiment lifetime. This peak-load phase is suitable for assessing the potential of an optimized preemptive CPU-dispatching scheme, and its actual advantages in the management of differentiated transaction priorities. The indication of peak-load has been evidenced by having the pool of contexts highly busy (above the 90%) for most of the experiments' duration. The reported performance results have been computed as the average over three repetitions of the experiment.

In Figure 5 we show the average turnaround time for transactions born at the 5 different priority levels. The turnaround time is computed as the sum of all the times spent by a transaction either for actual processing activities or while being kept within the priority queue—either as a cold or a hot transaction. Also, if a transaction is aborted and then retried, any aborted transaction run contributes to the turnaround time of the transaction. The baseline plot refers to a scenario where the STM does not use extra-ticks and, consequently, does not use preemption. Therefore, in the baseline configuration, a thread passes control to a standing higher priority transactional request only at the end of the processing phase of the currently executed transaction. For completeness of the analysis we also considered a setting where the extra-tick logic is active, but no-preemption is ever actuated. This configuration is useful for the assessment of the overhead caused by the extra-tick logic compared to the baseline case. Also, the preemptive STM architecture we have presented has been assessed by considering different settings for the value of C_{max}, and by either including or excluding the lazy priority promoting scheme for the management of the dynamic priority of the transactions (see Section III-C). By the results we see how, compared to the baseline, the preemptive approach reduces the average turnaround time of transactions born at higher priority levels (say levels 4 and 5) by around 60%-65%. Also, transactions at middle priority levels (e.g., level 3) exhibit an average turnaround latency essentially not penalized by preemption, or even slightly favored, while transactions born at lower priority levels (i.e., 1 and 2) show a penalization of their average turnaround which is mostly limited to less than 5%, and no more than 15% in the worst case. As expected, the higher advantages for higher priority transactions are achieved with larger values of C_{max}, which lead to delaying the dynamic increment of the priority of transactions born

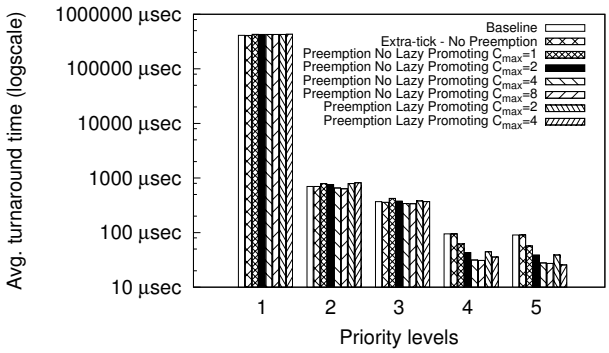

Figure 5. Average turnaround time for transactions born at different priority levels (log-scale on the y-axis).

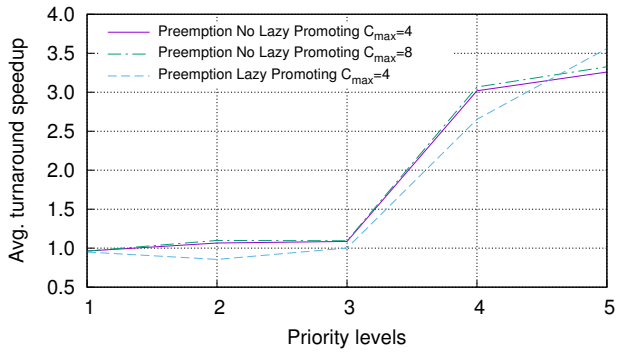

Figure 6. Speedup - Ratio between the turnaround time of the baseline configuration and the turnaround time of the preemptive configuration.

at low priority levels (e.g., level 1). The configuration where the extra-tick is active, but no preemption is ever actuated, shows performance essentially aligned with the one of the baseline, indicating negligible overhead of the extra-tick management logic.

In order to better outline the effects by the preemptive approach, we report in Figure 6 the ratio between the average turnaround latency provided by the baseline and the one provided by the preemptive approach, namely the speedup on the turnaround provided by the preemptive solution. For this plot we show the most promising configurations of the preemptive solution, selected on the basis of the results shown in Figure 5. The best configurations are still the ones with lager values of C_{max} (namely 4 or 8) and the plots show the effectiveness of our preemptive approach in both lazy promoting and no-lazy promoting scenarios. The configuration based on lazy promoting and C_{max} set to the value 4 is able to provide higher speedup (vs the baseline) compared to the one not employing lazy promoting for transactions born at priority level 5, at the expense of a reduction of the speedup for transactions born at priority level 2. This phenomenon is clearly due to the fact that, with lazy promoting, transactions born at priority level 1 dynamically acquire higher priority (e.g., 2) right after the first preemption, thus interfering more with transactions originally born at priority level 2. This phenomenon does not appear when lazy promoting is excluded.

Finally, in Figure 7 we report data indicating how the probability of abort varies in the different configurations. As stated before (see Section III-C), this variation can be caused by the effects of preemptions on the length of the vulnerability window of the transactions. By the results we see that transactions born at priority level 2 are those more impacted by this phenomenon. In particular, they show an increase of the abort probability—with consequent need for retries that lead to stretch the turnaround latency—for lower values of C_{max} and/or when lazy promoting is employed. As discussed before, this is caused by the higher interference caused by transactions born at priority level

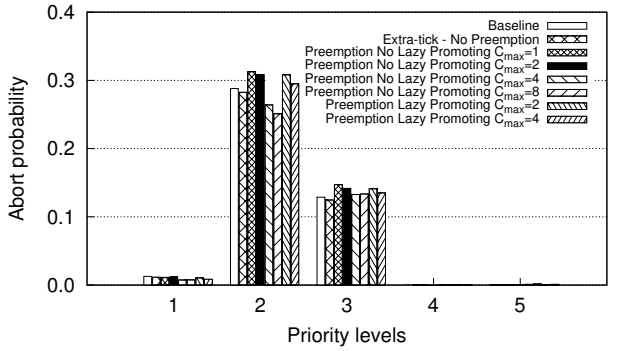

Figure 7. Variation of the transaction abort probability.

1, which dynamically acquire higher priority leading to increased concurrency between shorter transactions born at priority level 2 and the definitely longer ones born at priority level 1. The opposite behavior, with a reduction of the abort probability of transactions born at priority level 2, is instead noted when running with larger values of C_{max} or when excluding lazy promotion.

Overall, the experimental data support the effectiveness of our preemptive approach in favoring the turnaround time of higher priority transactions, compared to a baseline scenario that manages priorities according to a non-preemptive scheme. Also, the system/kernel level support we have employed for handling preemptions has been shown to induce negligible overhead, which further favors our solution.

V. Conclusions

In this article we have presented a preemptive Software Transactional Memory (STM) environment, where fine grain timer-interrupts—of the order of tens of microseconds—are delivered to the STM layer in order to enable a thread running some in-memory transaction to be promptly interrupted and to pass control to some standing higher priority transactional task. To the best of our knowledge this is the first attempt to provide such preemptive capabilities within

an STM environment, since state-of-the-art STM implementations CPU-dispatch a higher priority transactional task only after the finalization of the current one, thus not reacting to the injection of higher priority tasks with the same level of promptness. We have also presented a policy for dynamically changing the priority of transactions—depending on the behavior they show along their lifetime—in order to optimize the final performance delivered by the preemptive STM environment. Finally, we have reported the results of an experimental study based on a port of the TPC-C benchmark to STM, demonstrating the ability of our proposal to reduce the turnaround time of higher priority transactions, while not significantly un-favoring lower priority ones. In this study the priorities are determined on the basis of the CPU demand by the different transaction profiles, with lower demanding ones having higher priorities, a classical approach aimed at favoring shortest jobs.

REFERENCES

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

[2] M. Castro, L. F. W. Goes, C. P. Ribeiro, M. Cole, M. Cintra, and J.-F. Mehaut. A machine learning-based approach for thread mapping on transactional memory applications. In *Proceedings of the 18th International Conference on High Performance Computing*, pages 1–10, 2011.

[3] P. di Sanzo, M. Sannicandro, B. Ciciani, and F. Quaglia. Markov chain-based adaptive scheduling in software transactional memory. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium*, pages 373–382, 2016.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, pages 194–208, 2006.

[5] D. Didona, N. Diegues, A. Kermarrec, R. Guerraoui, R. Neves, and P. Romano. Proteustm: Abstraction meets performance in transactional memory. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 757–771, 2016.

[6] D. Didona, P. Felber, D. Harmanci, P. Romano, and J. Schenker. Identifying the optimal level of parallelism in transactional memory applications. *Computing*, 97(9):939–959, 2015.

[7] N. Diegues, P. Romano, and S. Garbatov. Seer: Probabilistic scheduling for hardware transactional memory. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures*, pages 224–233, 2015.

[8] S. Dolev, D. Hendler, and A. Suissa. Car-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of the 27th ACM symposium on Principles of Distributed Computing*, pages 125–134, 2008.

[9] A. Dragojević and R. Guerraoui. Predicting the scalability of an stm: A pragmatic approach. In *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, 2010.

[10] R. Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Research Cambridge Tech Report, Jan 2006.

[11] 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.

[12] M. Harchol-Balter, B. Schroeder, N. Bansal, and M. Agrawal. Size-based scheduling to improve web performance. *ACM Trans. Comput. Syst.*, 21(2):207–233, May 2003.

[13] W. Maldonado, P. Marlier, P. Felber, J. Lawall, G. Muller, and E. Rivière. Supporting time-based QoS requirements in software transactional memory. *ACM Trans. Parallel Comput.*, 2(2):10:1–10:30, July 2015.

[14] A. Pellegrini and F. Quaglia. Time-sharing time warp via lightweight operating system support. In *Proceedings of the 3rd ACM-SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 47–58, 2015.

[15] D. Rughetti, P. Di Sanzo, B. Ciciani, and F. Quaglia. 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*, pages 278–285, 2012.

[16] D. Rughetti, P. di Sanzo, B. Ciciani, and F. Quaglia. Analytical/ML mixed approach for concurrency regulation in software transactional memory. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 81–91, 2014.

[17] D. Rughetti, Paolo, F. Quaglia, and B. Ciciani. Automatic tuning of the parallelism degree in hardware transactional memory. In *Proceedings of the 20th International Conference Parallel Processing*, pages 475–486, 2014.

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

[19] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley Publishing, 8th edition, 2008.

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

[21] Q. Wang, S. Kulkarni, J. V. Cavazos, and M. Spear. Towards applying machine learning to adaptive transactional memory. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

[22] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 169–178, 2008.