Analysis of Polka Contention Manager for use in Multicore Hard Real-Time Systems

Adrien Quillet
adrien.quillet@ls2n.fr
LS2N, UMR CNRS 6004
Ecole Centrale de Nantes
Nantes, France

Audrey Queudet
audrey.queudet@ls2n.fr
LS2N, UMR CNRS 6004
University of Nantes
Nantes, France

Didier Lime
didier.lime@ls2n.fr
LS2N, UMR CNRS 6004
Ecole Centrale de Nantes
Nantes, France

ABSTRACT

Transactional memory (TM) draws the attention of both academic and development groups; indeed this concept offers an alternative to lock-based approaches, easing programmers' work. Despite the large amount of investigations around this topic, the question of the correctness of most TM implementations remains open. More specifically, the lack of upper bounds on the execution time of transactions prevents the use of TM in real-time systems. To address this issue, we introduce new realistic assumptions relative to real-time systems, which allow to ensure *wait-freedom* guarantees progress (i.e. all transactions progress) when Polka contention manager is considered. In that context, through a thorough formalization of the system, we prove upper bounds both on the number of abortions and on the execution time of transactions.

CCS CONCEPTS

• **Computer systems organization** → **Real-time operating systems**; *Multicore architectures*.

KEYWORDS

real-time systems, non-blocking synchronisation, software transactional memory

ACM Reference Format:
Adrien Quillet, Audrey Queudet, and Didier Lime. 2020. Analysis of Polka Contention Manager for use in Multicore Hard Real-Time Systems. In *Proceedings of RTNS '20: The 28th International Conference on Real-Time Networks and Systems (RTNS '20).* ACM, New York, NY, USA, 11 pages. https://doi.org/?

1 INTRODUCTION AND MOTIVATION

Real-time systems are increasingly used in electronic devices to control physical events. Those systems are designed to satisfy three objectives: *(i) logical determinism*, so runs with the same inputs always produce the same results, *(ii) temporal determinism* which guarantees that temporal constraints of every system action will be satisfied, and *(iii) reliability* which ensures the system is always available.

In the environment in which real-time systems operate, events usually take place concurrently. In order to handle them properly (i.e. to guarantee that all timing constraints will be satisfied), they resort to concurrent programming: each monitored event is managed by a processing entity called *task*. As multiple tasks can run in parallel over different processor cores, the underlying platform on which the system executes must provide both synchronization and communication mechanisms. In particular, an important aspect is the guarantee of data consistency. Resources shared between several tasks typically have a limit on the number of simultaneous accesses; as a consequence it is possible that a task can not progress due to an unavailable resource. The system must then efficiently manage resources to avoid both deadlocks and starvation while ensuring all the tasks still meet their deadlines.

One common way to protect concurrent modifications to shared memory is to use locks which are well known to be subject to priority inversion and deadlocks. Considering multicore platforms, another issue inherent to the use of such blocking mechanisms is that parallelism can be severely impacted since concurrent tasks competing for the same lock can be blocked, thus reintroducing some sequential execution in the parallel application. Recently [21], spin locks were studied for parallel real-time tasks in where each parallel task is scheduled exclusively on several pre-assigned processors (i.e., by the federated scheduling approach). This reduces the need for locks or scheduling to synchronize concurrent access (through shared resources may introduce locking as considered by Dinh et al. [5]). The authors developed new schedulability analysis techniques for parallel tasks with spin locks, and analysed blocking times but the analysis is pessimistic.

Another way to manage shared resources is to use the concept of transactional memory [16], which aims to significantly ease development and maintenance of concurrent programs. Transactional memory implementations exist both in hardware (*HTM - Hardware Transactional Memory*), in software (*STM - Software Transactional Memory*), or under schemes that combine both hardware and software (*Hybrid TM - Hybrid Transactional Memory*). In HTM systems, transactional memory support is implemented by modifying the data-cache protocols to support version management and conflict detection. The close synergy of the hardware with the processor

RTNS '20, June 09–11, 2020, Paris, France

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/?

core and cache allow these systems to provide very high levels of performance. Nevertheless, this type of TM system is usually bound by some sort of capacity contraints, e.g. the hardware can handle a specific fixed read-/write-set size. In contrast, STMs allow to build very flexible prototypes, as the full concurrency control is implemented in software. In return, STMs have a reputation for being slower than HTMs. However, considering the conceptual benefits of STMs, we will focus only on STMs in this paper. They allow programmers to embed a sequence of operations applied to shared resources into a transaction. Every transaction accesses shared data in memory without interfering with other transactions, and appears to them as executed atomically. To deal with inconsistencies, transactional memories usually use contention managers (CM) to resolve conflicts between transactions. Contention managers allow some transactions *to commit* (i.e. the data modifications are then made permanent), and force other conflicting ones *to abort* (i.e. all the modifications are discarded and the transaction restarts from the beginning), based on their internal policy.

Because software transactional memories hide to developers the way atomicity is obtained, it is essential to prove the correctness of their mechanisms, especially when used in real-time systems. In this paper, we introduce new assumptions on the execution time of transactions. We formally prove that the use of the *lock-free* contention manager Polka [24] in such constrained systems is sufficient to guarantee that every transaction progresses, bringing in upper bounds on the execution time and the number of abortions of a transaction.

The rest of this paper is organized as follows. We first present related works about transactional memories in Section 2. We then recall the main concepts of STM in Section 3. Section 4 introduces the formalization of the system we study. Section 5 focuses on Polka contention manager and contains formal proofs about its progress property. Finally, Section 6 concludes the paper.

2 RELATED WORK

2.1 Transactional Memory

Herlihy *et al.* were the first to propose a hardware based transactional synchronization methodology [16] that uses the cache coherence protocol to detect conflicts. Since then, the research community has produced many different HTMs [4, 13, 14, 19] that are more efficient. Chip vendors have increasingly begun to integrate HTM in their processors like the AMD's Advanced Synchronization Facility (ASF) [17]. Transactional synchronization extensions (TSX) whose performance was recently analysed in [22] were also adopted by Intel for its processors in the year 2012. However, since HTMs require hardware facilities, they are not suitable for most of multicore processors. Other software-based implementations using common atomic instructions such as CAS (Compare-and-Swap) or LL/SC (Load Linked/Store Conditional) [20], were first proposed by *Shavit and Touitou* in [26]. By nature, they are less efficient than HTM but they can be used with all multicore processors.

2.2 TM Correctness and Timing Analysis

The first steps to formally verify TM systems were made through *Greedy* [10] and *FTGreedy* [9] CM proposals. Guerraoui *et al.* proposed a new and robust CM that provides provable properties (e.g. *every transaction commits within bounded time*). They also introduced the notion of *opacity* [11], a correctness criterion for TM implementations.This idea was extended by Imbs *et al.* in [18]. Schoeberl et al. proposed *Real-Time Transaction Memory (RTTM)* [25], a new hardware-implemented synchronization paradigm for hard real-time systems. Their demonstrate that, provided the transaction conflict resolve time is bounded, tasks will meet their deadlines. Nevertheless, they do not provide any guarantees that transactions will complete successfully.

Sarni *et al.* presented Real-Time STM [23], a TM protocol ensuring *lock-freedom*, a weaker property than *wait-freedom*, that does not guarantee that *all* transactions will meet their deadlines, thus being suitable only for soft real-time systems.

Upper bounds for the number of transaction retries in a real-time system were provided by Schoeberl *et al.* [25] and El-Shambakey *et al.* [6, 7]. However, the conditions were very simple, with periodic tasks and one transaction performed by each task.

Cotard *et al.* [3] proposed a new STM for hard real-time systems, that relies on a wait-free protocol that enforces progress of every task and that can be integrated in the WCET of tasks during the timing analysis of the system. However, the approach is restricted to transactions that manipulate objects in a single access mode, either read or write.

Barros *et al.* [1] addressed the response time analysis of hard real-time tasks, which share STM data under partitioned scheduling strategies. They assume that transactions are serialised according to their arrival time and they are managed by following a FIFO contention manager for real-time systems (FIFO-CRT) [2].

Although those works are close in purpose to ours, none of them prove the *wait-freedom* property (i.e. all trans- actions progress) among existing CMs.

3 BACKGROUND MATERIAL

3.1 Software Transactional Memory Concepts

A transaction, as originally formalized in [8], is based on the notion of synchronization point, which represents a durable and consistent state of a computer system. This notion relies on the four ACID properties [12]:

- *Atomicity.* The sequence of operations either all occur (the transaction *commits*), or nothing occur (the transaction *aborts*), which makes the sequence of operations indivisible.
- *Consistency.* The transaction starts and ends in a valid system state.
- *Isolation.* A transaction being executed is unaware of other executing transactions.
- *Durability.* Once a transaction has been committed, it will remain so.

During its execution, a transaction will manipulate one or several data, either in reading or writing modes. This set of data is called the *transaction dataset*.

Definition 3.1. Two transactions are said *in contention* if they both access a same resource during their execution and one of them may update it.

Definition 3.2. Two transactions are said *in conflict* if they performed an operation on the same resource and one of them updated it.

An STM needs a *data version management policy* in order to maintain old values, needed to abort the transaction, and new values, needed to commit the transaction. Concurrency control is based on a *conflict detection policy* which defines the rules triggering the abortion of a transaction. There are two approaches: *(i)* with the *eager* approach, conflicts are detected and resolved as soon as they occur, *(ii)* with the *lazy* approach, conflicts detection and resolution occur later, generally before trying to commit the transaction. Finally, a transactional memory typically has a contention manager module [15], which is in charge of resolving conflicts between transactions. Depending on its contention policy (see [24] for a survey), the CM decides which of the conflicting transactions can continue, and as a result, which transaction(s) must abort.

3.2 Polka Contention Manager

The Polka contention manager [24] favors the transactions which performed the greater number of operations. The chosen criterion to quantify how much work has been done at a given time is the number of data already accessed either in reading or writing mode; it is called *karma* and is denoted by k. It is a component of the state of the transaction. Transactions start with a zero karma which is incremented by one at each first access to a data (first access is renewed each time the transaction starts or restarts its operations) or at each abortion of the transaction. When a transaction aborts, the maximum number of times it may restart is bounded by the difference in karma between itself and its enemy. Moreover, Polka introduces an exponential backoff between successive transaction restarts. Every aborted transaction waits for a random duration that increases exponentially. When a transaction commits, its karma returns to zero.

4 MODELS AND TERMINOLOGY

4.1 System Model

We consider a platform made of m identical processor cores $\pi = \{\pi_1, \ldots, \pi_m\}$, so the maximum number of task instances that can run in parallel is m.

4.1.1 Time Representation. The system time sequence $t = t_{i1 \leq i}$ is a sequence of countable instants. We denote by $\mathbb{T}$ the domain of t. At any time, the difference between two subsequent instants t_{i+1} and t_i is 1; this minimum period is called a *time unit*.

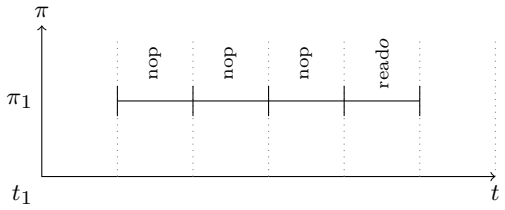

Figure 1: Execution model for a 4 time units operation

4.1.2 Objects. We assume that every shared resource managed by an STM is actually manipulated with *an object*. We denote by $O_{\mathcal{S}}$ the finite set of system objects such that $O_{\mathcal{S}} = O_{\overline{\Sigma}} \cup O_{\Sigma}$, where $O_{\overline{\Sigma}}$ and O_{Σ} denote the finite set of objects used only outside (resp. inside) an explicit transaction. The final set of all possible values for system objects is denoted by $\Upsilon^{\perp} = \Upsilon \cup \{\perp\}$ where Υ represents the finite set of all possible defined values for the objects whereas $\perp$ stands for the undefined value. The value of objects is given by a function $\mathrm{Val}_{\mathcal{S}} : O_{\mathcal{S}} \rightarrow \Upsilon^{\perp}$ such that:

$$\mathrm{Val}_{\mathcal{S}} o = x \text{ iff x is the value of object o}$$

4.1.3 Operations. Let $\Theta_{\mathcal{S}}$ denote the finite set of system operations. We consider three possible operations:

- read$o$: $\forall o \in O_{\mathcal{S}}$, it returns the value of object o (i.e. $\mathrm{Val}_{\mathcal{S}} o$);
- writeo, x: $\forall o \in O_{\mathcal{S}}, x \in \Upsilon^{\perp}$, it writes the value x in object o;
- nop: it stands for the *idempotent operation*, thus leaving the system in the same state as it was before.

We assume that each operation requires one time unit to be performed. Note that every other kind of operations (e.g. arithmetic ones) can be abstracted to a sequence of *read()*, *write()* and *nop()* operations. In order to model an operation that takes more than one time unit to execute, the operation has to be preceded by *nop()* operations. As a result, to model an operation that requires k time units, it must be preceded by $k-1$ idempotent operations, as illustrated in Figure 1.

4.2 Task Model

Let $T_{\mathcal{S}}$ denote the finite set of system tasks. A task τ_i is a tuple $\langle \pi_i, \phi_i, T_i, \Sigma_i \rangle$, where π_i is the processor core on which all *instances* (or *jobs*) of τ_i will be executed, ϕ_i is the offset of first activation of the task, T_i is the minimum period between two activations of the task and Σ_i is the finite set of transactions launched during every job.

Each system task τ_i produces an infinite number of successive jobs $\mathrm{J}_{i,j}$, with $j \in \mathbb{N}^*$, as illustrated in Figure 2. All jobs of a given task are assumed independent. Job $\mathrm{J}_{i,j}$ corresponds to the j^{th} activation of task τ_i. We denote by $r_{i,j} \geq r_{i,j-1} + T_i$ the release time of job $\mathrm{J}_{i,j}$. The release time $r_{i,0}$ of the first job of task τ_i is equal to ϕ_i.

Definition 4.1. A job is said *active* as soon as released.

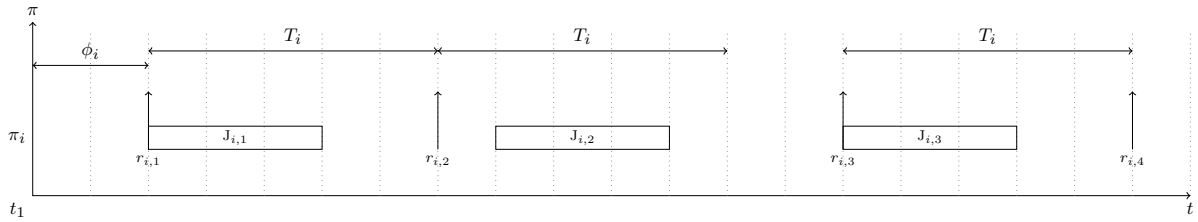

Figure 2: Execution of a task $\tau_i = \langle \pi_i, \phi_i, T_i, \Sigma_i \rangle$

At any time, there exists at most one active job for a given task. An active job can be preempted at some time t_n, except if it is executing a transaction.

4.3 Transaction Model

First, we assume that a transaction only belongs to one task.

4.3.1 Formalization. A transaction σ_i is a tuple $\langle b_i, N_i, \Omega_i, O_i \rangle$, where b_i represents its start time, N_i is the number of operations in the transaction, Ω_i is its dataset with $\Omega_i \subseteq O_\Sigma$ and O_i is the transaction private memory, which contains copies of the objects used by the transaction. This private memory is only visible to the transaction itself. Hence, a transaction works on its own copies of the objects and if it successfully commits, then the commit phase is completed by copying the private modifications into the shared memory.

We denote by $\eta_i = \langle \xi_i, n_i, \omega_i, c_i, \mathrm{Val}_i \rangle$ the state of σ_i, where ξ_i is the current status of σ_i, n_i is the actual number of performed operations, ω_i is its finite set of already opened objects with $\omega_i \subseteq \Omega_i$, c_i is its internal clock starting at 0 and increased by one at each performed operation, and Val_i from $O_i \to \Upsilon^\perp$, is the function giving the value of an object from its transaction private memory. There exists 5 different transaction status:

- *NotStarted* is the status of a transaction that has never started;
- *Running* is the status of a processing transaction which has at least one more operation to perform;
- *Checking* is the status of a transaction which performed all its operations, and is verifying that its modifications will not leave the system in an inconsistent state;
- *Aborted* is the status of a transaction which has been aborted;
- *Committed* is the status of a transaction that has completed its execution. All committed modifications are visible to the entire system.

Definition 4.2. A transaction is said *processing* if its status is either *Running*, *Checking* or *Aborted*.

Definition 4.3. A transaction is said *active* if its status is either *Running* or *Checking*.

We denote by $\mathrm{Exec}_\Sigma t_n$ the set of processing transactions at time t_n, and by $\mathrm{Active}_\Sigma t_n$ the set of active transaction the same time.

We denote by Σ the finite set of system transactions and by $\mathcal{N} = \{\eta_1, \ldots, \eta_{|\Sigma|}\}$ the finite set of transaction states.

Let Θ_Σ denote the finite set of operations specific to transactions, such that it exists 5 different operations related to transactions:

- *start*: starts transaction σ_i, initializing all its parameters properly.
- *restart*: restarts transaction σ_i after an abortion. This operation resets all transaction parameters except its start time which remains unchanged.
- *check*: brings transaction σ_i into checking phase. This operation checks if all previously opened objects keep the same values, thus detecting potential conflicts with other transactions.
- *abort*: cancels all modifications made inside the transaction itself.
- *commit*: makes all modifications made by the transaction visible for the entire system.

Note that depending on other currently executed transactions and on the read-/write-set sizes, the execution time of *check*, *abort*, and *commit* operations may vary.

Transactions can also perform any operation of $\Theta_\mathcal{S}$. Accordingly, the domain of arguments of system operations is extended to $O_\mathcal{S} \cup O_1 \cup \ldots O_{|\Sigma|}$. The set of operations that is possible to perform from a given status is illustrated in Figure 3.

4.3.2 Properties. The system state resulting of the parallel execution of transactions is consistent only if this execution is serializable.

Definition 4.4. A parallel execution of n transactions is *serializable* if it is equivalent to a serial execution of these n transactions.

We propose to distinguish the following notion of transaction correctness.

Let *TT (Turnaround Time)* denote the upper bound on the time during which transactions can remain active in the system. This bound is set *a priori* by programmers. For instance, considering a real-time application, this upper bound may be set according to the real-time constraints of the tasks, using a worst case execution time (WCET) analysis.

Definition 4.5. A transaction is said *timely correct* if the maximum period between its (*start()*|*restart()*) and (*abort()* | *commit()*) operations is less than or equal to TT. Otherwise (i.e. this period is exceeded), the transaction is said *timely incorrect*.

Since a transaction has to perform at least one system operation other than *nop()* and three specific operations (*start()*, *check()* and *commit()*) to be valid, the minimum value of TT is 4 time units.

Let $\sigma_i \dagger \sigma_j$ (resp. $\sigma_i \ddagger \sigma_j$) denote the fact that transactions σ_i and σ_j are in contention (resp. in conflict). By extension, we denote by $\sigma_i \dagger \text{Enemies}_i$ (resp. $\sigma_i \ddagger \text{Enemies}_i$) the fact that σ_i is in contention (resp. in conflict) with every transaction of the transaction set Enemies_i.

Definition 4.6. The sequence of operations performed between (*start()*|*restart()*) and (*abort()*|*commit()*) operations (included) of a transaction is called *cycle of execution* of this transaction.

Definition 4.7. A transaction σ_i *progresses* if, at a given time t_n, $\sigma_i \in \text{Active}_\Sigma t_n$ and there exists $t_m > t_n$ such as at time t_m, the status of σ_i is *Committed*.

For ease of reference, notations previously presented are summarized in Table 1.

Symbol	Description
π	Core set of the system
$T_\mathcal{S}$	Task set of the system
Σ	Transaction set of the system
$O_\mathcal{S}$	Object set of the system
O_Σ	STM-managed object set
$O_{\overline{\Sigma}}$	Non-STM-managed object set
π_m	Core m of the core set π
τ_i	Task i of the task set $T_\mathcal{S}$
$\text{J}_{i,j}$	J^{th} job of task i
σ_k	Transaction k of the transaction set Σ
O_k	Private memory of transaction k
Ω_k	Dataset of transaction k
b_k	Start date of transaction k
ξ_k	Status of transaction k
k_k	Karma of transaction k
t_n	Time instant n
$\text{Active}_\Sigma t_n$	Set of active transactions at time t_n
$\text{Exec}_\Sigma t_n$	Set of processing transactions at time t_n
TT	Turnaround Time

Table 1: Notations used in this paper

5 SCOPING POLKA CM

In this section, we discuss how the *karma* values assigned by Polka CM to transactions may vary along time. In particular, we study the extreme cases for which transactions exhibit minimal or maximal *karma* values.

Let us first recall how conflicts are resolved in Polka CM. So, let us assume that at a given time there exists both a checking transaction σ_i and a set Enemies_i of transactions conflicting with σ_i. The following two cases are therefore possible:

- $\forall \sigma_e \in \text{Enemies}_i : k_i > k_e$: every enemy transaction aborts and σ_i commits,
- $\exists \sigma_e \in \text{Enemies}_i : k_i \leq k_e$: σ_i sleeps for Δ_w time units. Meanwhile, if a transaction from Enemies_i commits then σ_i aborts, and if every transaction from Enemies_i aborts, then σ_i commits. At the end of its sleeping period, if σ_e is still active, then σ_i sleeps again only if its maximum number of sleeping periods $w_{i,e}^{\max} = k_e - k_i$ is not reached. The sleeping time Δ_w is randomly drawn from an interval growing exponentially with each sleeping period, such that $\Delta_w \in 1, 2^{w_{i,e}}$, where $w_{i,e}$ is the number of achieved sleeping periods. If $w_{i,e} \geq w_{i,e}^{\max}$ and σ_e is still active, then σ_e aborts and σ_i commits.

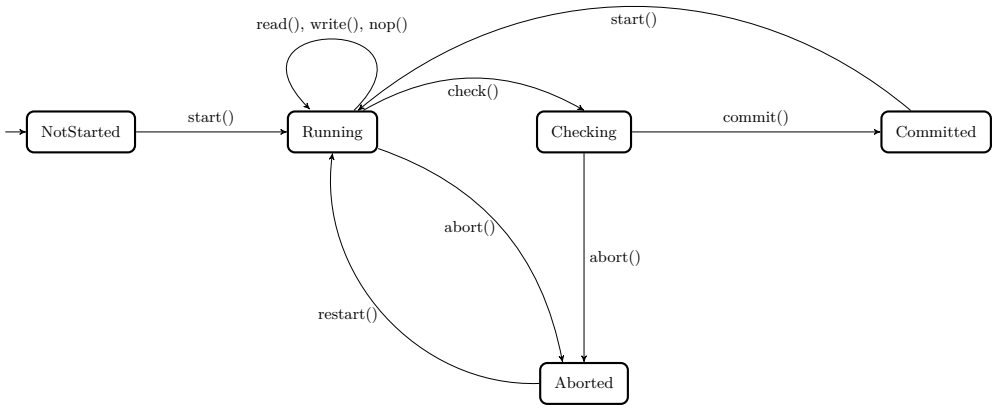

Figure 3: State-transition diagram of possible transaction status

5.1 Illustrative Example

Consider a set of $(n+1)$ transactions, with n slow readers and one fast writer. The execution scenario is illustrated in Figure 4. Transaction σ_w writes into object o and n transactions read the value of this object.

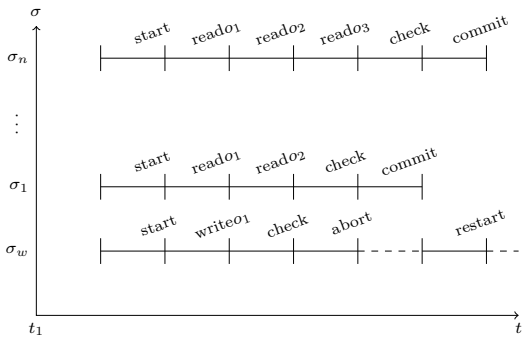

Figure 4: Fast single-writer/Slow multiple-reader scenario

Considering lazy conflict detection (*check()* operation occurs right before *commit()* operation), we can deduce the two following commit rules:

- If the writer commits, then all readers whose *read()* operation is performed abort;
- If a reader commits, then all readers commit and the writer aborts if its *write()* operation is already performed.

Let's investigate Polka arbitration among competing transactions. All transactions earn one karma unit for their execution, and one more karma unit if they abort. At the beginning of the execution, all karmas are equal to zero; the writer will commit first because it runs faster than readers. Its karma will then return to zero while the karma of all readers will be of 2 (one for the performed operation, and one for the abortion). The writer is a priori favored since it reaches validation before any reader, but Polka contention manager favors transactions that have opened the greater number of objects, their karma being incremented by one at each first object opening. As a result, if the writer's karma is lower than the karma of readers, the writer will sleep, waiting for readers to commit (see Figure 4). The only chance for readers to validate is if the writer total sleeping period is strictly greater than the difference of karma between readers and writer. If so, readers commit successfully while the writer still sleeps; the writer is aborted, its karma is increased by one and the karma of all readers return to zero.

The sleeping period being randomly drawn from an interval growing exponentially, the more readers are aborted, the more likely they are to validate, because the writer sleeping period has a chance to exponentially grow.

Without studying more specifically Polka policy, we cannot say if readers will eventually validate. They have more and more chances to commit, but if the total sleeping period of the writer indefinitely remains lower than the difference of karma between the readers and the writer, then the writer will indefinitely abort all readers. Hereafter, we focus on the worst (i.e. slowest) and best (i.e. fastest) karma accumulation for transactions in terms of speed.

5.2 Slowest Karma Accumulation

Lemma 5.1. *In a system using a software transactional memory whose contention manager is Polka, the slowest karma accumulation at time t_n of a timely correct transaction σ_l is given by:*

$$k_l = \lfloor \frac{t_n - b_l}{TT} \rfloor \times 2 + \epsilon_l \text{ with } \epsilon_l \in \{0, 1\}$$

Proof. Let σ_l denote the timely correct transaction whose karma increases the slowest. According to Polka policy, the slowest way to increase karma is to manipulate the fewest possible objects, while taking the maximum amount of time to do it. Since we study karma accumulation, it is assumed that this transaction indefinitely aborts. Hence σ_l must have a single operation to perform (so its dataset contains only one object). The timing diagram for the execution of σ_l is illustrated in Figure 5.

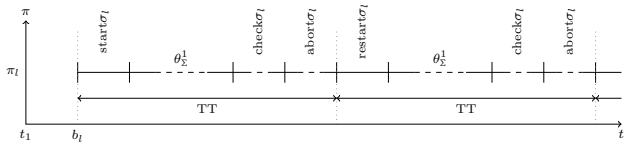

Figure 5: Slowest karma accumulation by a transaction σ_l

Transaction σ_l increases its karma by one not only when it accesses its single object but also at each *abort()* operation. As a result, its karma is increased by 2 every TT time units. So, at given time, the karma of σ_l is double the number of execution cycles of the transaction, in addition to the karma ϵ_l accumulated during the current execution cycle. □

5.3 Fastest Karma Accumulation

Lemma 5.2. *In a system using a software transactional memory whose contention manager is Polka, whose conflict detection is lazy and in which all transactions are timely correct, the fastest karma accumulation at time t_n of transaction σ_w is given by:*

$$k_w = \lfloor \frac{t_n - b_w}{TT} \rfloor \times \Big(TT - 2 \Big) + \epsilon_w$$
$$\text{with } \epsilon_w \in \{0, \ldots, n\}$$

Proof. Let σ_w denote the timely correct transaction whose karma increases the fastest. According to Polka policy, the fastest way to accumulate karma is to access as many various objects as possible in a minimum amount of time. Because *(start()/restart())* and *check()* operations do not increase karma, a timely correct transaction accessing as many objects as possible in TT time units eventually has a greater karma than a transaction accessing a single object in one time unit that is aborted and restarted over a period of TT time

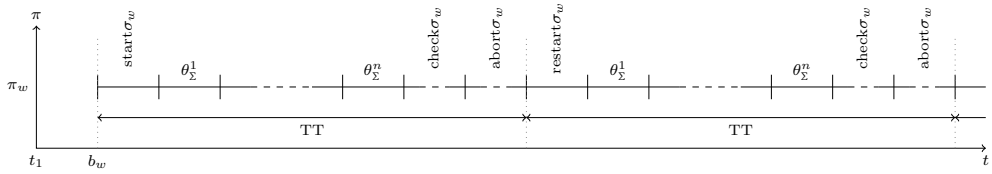

Figure 6: Fastest karma accumulation by a transaction σ_w

units. As previously, it is also assumed that this transaction indefinitely aborts.

Transaction σ_w thus accesses to a maximum number of objects over its interval of correctness, keeping the remaining time units to perform operations (*start()/restart()*), *check()* and (*abort()*). The timing diagram for the execution of σ_w is illustrated in Figure 6.

Transaction σ_w increases its karma by one for each performed operation or *abort()* operation. Consequently, its karma is increased by $\big(\mathrm{TT}-2\big)$ every TT time units. So, at a given time, the karma of σ_w is $\big(\mathrm{TT}-2\big)$ times the number of execution cycles of the transaction, in addition to the karma ϵ_w accumulated during the current execution cycle.

□

5.4 Achieving the Greatest Karma

LEMMA 5.3. *In a system using a software transactional memory whose contention manager is Polka, whose conflict detection is* lazy *and in which all transactions are timely correct, the maximum possible karma value is*

$$k_{\mathcal{S}}^{max} = \Big(min\big(|\pi|,|\Sigma|\big)-1\Big)\times\Big(TT-2\Big)+\epsilon$$
$$with\ \epsilon\in\{0,\ldots,n\}$$

PROOF. Let us argue by contradiction, assuming that there is no upper bound on the karma.

First, let us study the trivial case where there is only one transaction in the system. A single transaction cannot be in conflict with itself and thus cannot be aborted. As a consequence, this single transaction always commits and always achieves the greatest karma which is the number of objects it accessed. Hence there exists an upper bound ϵ on the karma equals to the number of objects opened by the transaction during its execution.

Let now consider that there are at least two timely correct transactions σ_w and $\sigma_{w'}$ in the system, such that there is no upper bound on their karma. Assuming that $k_i t_z$ denote the karma of a transaction σ_i at time t_z, this can be expressed by the following property:

$$\forall t_n, \exists t_m > t_n :$$
$$\Big(k_w t_n < k_w t_m\Big) \vee \Big(k_{w'} t_n < k_{w'} t_m\Big) \quad (1)$$

Let us prove this property is false.

The lack of upper bound on the karma means that there is no upper bound on the number of abortions of a transaction. As a consequence, there always exists a transaction which karma is greater than the previous greatest karma value. Note that the transaction with the greatest karma can vary over time. To examine the evolution of karma, we consider the worst-case execution for system transactions. We assume that all transactions accumulate karma the fastest way and share the same dataset; the timing diagram of their execution is illustrated in Figure 6. To maximize contention between transactions, we also consider the execution case illustrated in Figure 7. The maximum number of concurrent transactions is the minimum between the number of system transactions and processor cores, and all transactions start at the same time.

Let σ_w denote the transaction executed on processor core π_w. Transaction σ_w reaches its checking phase before any

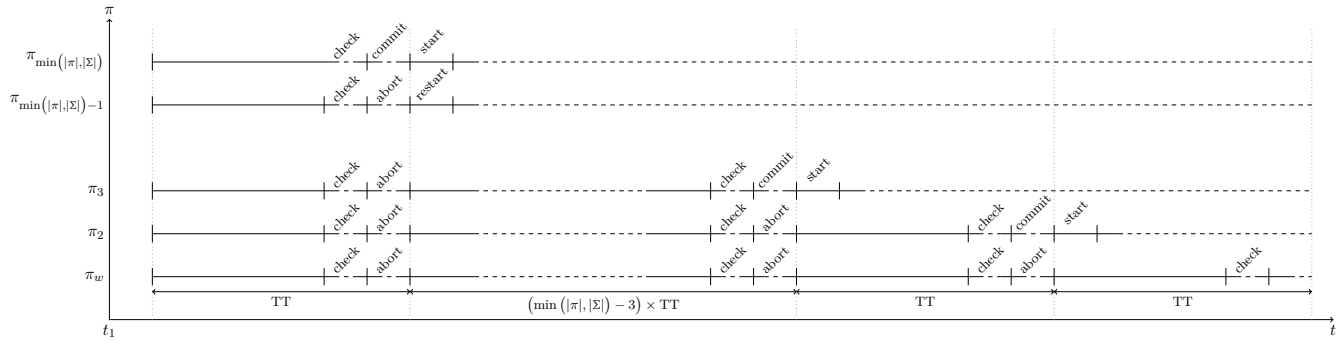

Figure 7: Worst-case contention case for a fast accumulation karma transaction σ_w

All transactions start at the same time and are based on the same model (as illustrated in Figure 6). As a result, if a transaction commits, it forces the abortion of all other transactions.

other transaction; all transactions having the same karma, σ_w has to sleep, waiting for other transactions to commit or abort. According to the execution case, all transactions are in conflict, so only one transaction will be allowed to commit. The last transaction reaching its checking phase will commit, and other transactions will abort, particularly σ_w. After a bounded number of abortions and always according to the execution pattern illustrated in Figure 7, σ_w will possess the greatest karma in the system. Let $\sigma_{w'}$ denote the transaction that possesses the second greatest karma in the system; it is the transaction that committed first in the system. According to Lemma 5.2, at time t_n, the karma of $\sigma_{w'}$ can be expressed as:

$$k_{w'} = \lfloor \frac{t_n - b_{w'}}{\mathrm{TT}} \rfloor \times \big(\mathrm{TT} - 2\big) + \epsilon_{w'}$$
$$\text{with } \epsilon_{w'} \in \{0, \ldots, n\}$$

while the karma of σ_w is:

$$k_w = \lfloor \frac{t_n - b_w}{\mathrm{TT}} \rfloor \times \big(\mathrm{TT} - 2\big) + \epsilon_w$$
$$\text{with } \epsilon_w \in \{0, \ldots, n\}$$

Since σ_w has executed one more execution cycle than $\sigma_{w'}$, and considering they have the same execution pattern, the karma of σ_w can also be expressed as:

$$\begin{aligned} k_w &= \lfloor \frac{t_n - b_{w'} - \mathrm{TT}}{\mathrm{TT}} \rfloor \times \big(\mathrm{TT} - 2\big) + \epsilon_{w'} \\ &= \big(\lfloor \frac{t_n - b_w}{\mathrm{TT}} \rfloor + 1\big) \times \big(\mathrm{TT} - 2\big) + \epsilon_w \end{aligned}$$

As a result, the maximal difference of karma between σ_w and $\sigma_{w'}$ is $\big(\mathrm{TT} - 2\big)$. We then distinguish two possible cases:

- *case (i):* $\sigma_{w'}$ tries to commit first.
- *case (ii):* σ_w tries to commit first.

case (i): If $\sigma_{w'}$ tries to commit first, then $\sigma_{w'}$ will sleep, waiting for σ_w, because the karma of σ_w is greater than its karma. The difference of karma between σ_w and $\sigma_{w'}$ being $\big(\mathrm{TT} - 2\big)$, $\sigma_{w'}$ will sleep at least $\big(\mathrm{TT} - 2\big)$ time units before aborting σ_w. Transaction σ_w being in checking phase too, and considering its execution pattern, $\sigma_{w'}$ only has one time unit to abort σ_w to commit, i.e. $\mathrm{TT} - 2 \leq 1$, that is equivalent to $\mathrm{TT} \leq 3$. By definition, $\mathrm{TT} \geq 4$. As a consequence, $\sigma_{w'}$ cannot commit before σ_w.

case (ii): If σ_w tries to commit first, then σ_w will commit because it has the greatest karma.

In both cases, the karma of σ_w at t_n allows it to commit. If σ_w can commit, its karma cannot indefinitely increase. We can deduce the same fact for other transactions since they all share the same execution pattern. As a consequence, there exists an upper bound on the karma. According to the execution pattern of transactions, the greatest karma is equal to the number of concurrent transactions (not counting the current transaction) times the karma accumulated during an execution cycle (i.e. $\big(\mathrm{TT} - 2\big)$) in addition to the karma accumulated during its last cycle (at most n). This contradicts Property 1. □

Lemma 5.4. *In a system using a software transactional memory whose contention manager is Polka, whose conflict detection is* lazy *and in which all transactions are timely correct, the maximum period needed by a recurring aborted transaction σ_l to reach the maximum possible karma $k_{\mathcal{S}}^{max}$ is given by:*

$$\Delta_l^{max} = \big(\lfloor \frac{k_{\mathcal{S}}^{max}}{2} \rfloor + 1\big) \times TT$$

Proof. According to Lemma 5.1, the slowest karma accumulation is 2 karma units every TT time units for a timely correct transaction that always aborts. According to Lemma 5.3, there exists an upper bound $k_{\mathcal{S}}^{\max}$ on the karma that is possible to accumulate in the system. As a consequence, there exists an upper bound on the time needed to reach this karma limit.

Assuming the worst-case execution, a timely correct transaction increases its karma by 2 every TT time units. As a consequence, after $\lfloor \frac{k_{\mathcal{S}}^{\max}}{2} \rfloor$ cycles of execution, the karma of this transaction is near or equal to $k_{\mathcal{S}}^{\max}$. Obviously, if it has not already happened, the transaction will reach the karma limit during its next cycle of execution. We obtain the upper bound established in the Lemma. □

5.5 Polka: a Wait-free CM

Theorem 5.5. *In a system using a software transactional memory, which contention manager is Polka, and which conflict detection is lazy, if every transaction is timely correct, then every transaction progresses.*

Proof. We assume that in such a system, there exists a timely correct transaction σ_l that never progresses, as expressed by the property:

$$\exists t_i : \sigma_l \in \mathrm{Activest}_i \wedge \forall t_j > t_i : \xi_l \neq \mathrm{Committed} \qquad (2)$$

Let us prove this property is false.

Let $\mathrm{Enemies}_l t_m$ denote the set of transactions in conflict with σ_l at time t_m. We need to study four different cases:

- $\xi_l = \mathrm{Checking} \wedge \sigma_l \ddagger \mathrm{Enemies}_l$
  (i.a) $\forall \sigma_e \in \mathrm{Enemies}_l : k_l > k_e$
  (i.b) $\exists \sigma_e \in \mathrm{Enemies}_l : k_l \leq k_e$
- $\xi_l = \mathrm{Running} \wedge \sigma_l \ddagger \mathrm{Enemies}_l$
  (ii.c) $\exists \sigma_e \in \mathrm{Enemies}_l : \xi_e = \mathrm{Checking} \wedge k_e > k_l$
  (ii.d) $\exists \sigma_e \in \mathrm{Enemies}_l : \xi_e = \mathrm{Checking} \wedge k_e \leq k_l$

(i.a). Transaction σ_l is in checking phase and its karma is strictly greater than the karma of every transaction conflicting with it. Transaction σ_l is allowed to commit, and all conflicting transactions are aborted. It contradicts Property 2.

(i.b). Transaction σ_l is in checking phase and there is at least one transaction $\sigma_e \in \mathrm{Enemies}_l$ having a karma greater or equal to σ_l's one. Transaction σ_l sleeps, waiting for σ_e. As a result, there are four possible cases:

(1) Transaction σ_e commits. Transaction σ_l aborts. Transaction σ_e karma returns to zero and σ_l karma is steadily increasing. According to Lemma 5.4, there exists a time

instant $t_n > t_m$ such that σ_l reaches the karma limit, i.e. $\forall \sigma_e \in \text{Enemies}_l t_n : k_l > k_e$. We refer to cases *i.a* and *ii.d*.

(2) Transaction σ_e aborts. There are again two possible cases:
- Transaction σ_l also aborts. A third transaction, in conflict with both σ_e and σ_l, commits. As a consequence, both transactions σ_e and σ_l abort. According to Lemma 5.4, there exists a time instant $t_n > t_m$ such that σ_l reaches the karma limit, i.e. $\forall \sigma_e \in \text{Enemies}_l t_n : k_l > k_e$. We refer to cases *i.a* and *ii.d*.
- Transaction σ_l is still active. Transaction σ_l is allowed to commit. It contradicts Property 2.

(3) $w_{l,e} < w_{l,e}^{max}$, i.e. the number of sleeping periods of σ_l, waiting for σ_e, is not greater than the difference between the karma of σ_l and σ_e. Transaction σ_l sleeps again, waiting for σ_e. We refer to cases *i.b.1*, *i.b.2* and *i.b.4*.

(4) $w_{l,e} = w_{l,e}^{max}$, the maximum number of sleeping periods is reached. Transaction σ_e aborts and transaction σ_l commits. It contradicts Property 2.

(ii.c). There is at least one checking transaction σ_e in conflict with σ_l having a karma strictly greater than σ_l's one. Transaction σ_l aborts and σ_e commits (see Figure 8). According to Lemma 5.4, there exists a time $t_n > t_m$ such that σ_l reaches the karma limit, i.e. $\forall \sigma_e \in \text{Enemies}_l t_n : k_l > k_e$. We refer to cases *i.a* and *ii.d*.

(ii.d). There is at least one checking transaction σ_e in conflict with σ_l having a karma lower or equal to σ_l's one. σ_e sleeps, waiting for σ_l. As a result, there are four possible cases:

(1) Transaction σ_l commits. Transaction σ_e aborts (see Figure 8). It contradicts Property 2.

(2) Transaction σ_l aborts. Transaction σ_e is allowed to commit if it is not aborted too. We refer to case *ii.d.4*.

(3) $w_{e,l} < w_{e,l}^{max}$, the number of sleeping periods is not greater than the difference between σ_e and σ_l karmas. Transaction σ_e sleeps again, waiting for σ_l (see Figure 9). We refer to cases *ii.d.1*, *ii.d.2* and *ii.d.4*.

(4) $w_{e,l} = w_{e,l}^{max}$, the maximum number of sleeping periods is reached. Transaction σ_l aborts and transaction σ_e commits (see Figure 9).
Let investigate this last case. By hypothesis, σ_l never progresses, that is in this precise situation, transaction σ_e always aborts σ_l by reaching the maximum number of sleeping periods. Without loss of generality, we assume the virtual worst-case execution for σ_l: this transaction accumulates karma the slowest way, as described by Lemma 5.1. Transaction σ_e accumulates karma the fastest way, as described by Lemma 5.2. Since both transactions are timely correct, their execution time, including sleeping periods, cannot be greater than TT. As a consequence, the maximum number of sleeping periods is TT. This is expressed by the following inequality:

$$k_l - k_e \leq \text{TT} \tag{3}$$

with

$$k_l = \lfloor \frac{t_n - b_l}{\text{TT}} \rfloor \times 2 + \epsilon_l$$
$$k_e = \lfloor \frac{t_n - b_e}{\text{TT}} \rfloor \times \big(\text{TT} - 2\big) + \epsilon_e$$

Since σ_e already aborted σ_l at least one time, it started right after σ_l, considering the worst-case execution. So σ_e karma can be expressed as:

$$\begin{aligned} k_e &= \lfloor \frac{t_n - b_l + 1}{\text{TT}} \rfloor \times \big(\text{TT} - 2\big) + \epsilon_e \\ &= \lfloor \frac{t_n - b_l}{\text{TT}} \rfloor \times \big(\text{TT} - 2\big) + C_e \end{aligned}$$

with C_e being a constant value which is equal to the sum of all constants in the previous inequality. The inequality 3 can be rewritten as:

$$\begin{aligned} \big(\lfloor \frac{t_n - b_l}{\text{TT}} \rfloor \times 2 + \epsilon_l\big) - & \\ \big(\lfloor \frac{t_n - b_l}{\text{TT}} \rfloor \times \big(\text{TT} - 2\big) + C_e\big) &\leq \text{TT} \\ t_n - b_l + \epsilon_l - C_e &\leq \text{TT} \\ t_n &\leq \text{TT} + b_l - \epsilon_l + C_e \\ t_n &\leq C \end{aligned}$$

with C being a constant value which is equal to the sum of all constants in the inequality. The system time sequence t being a strictly increasing sequence without supremum, the inequality 3 is incorrect. As a consequence, there is no transaction σ_e such that σ_e always aborts σ_l. It contradicts the Property 2.

□

6 CONCLUSION AND FUTURE WORK

While software transactional memory performances are widely studied, the question of the correctness of such mechanisms remains open. The few papers that investigated this issue actually focused on the design of new contention managers and on the modeling of transactions themselves, while there exists STM implementations with empirically efficient CMs. Motivated by this observation, we introduced new assumptions on the execution time of transactions into real-time systems in order to formally prove that, provided these systems use the existing Polka CM, they can ensure the progress of all transactions. Based on an fine-grain modeling of the whole system, we also derived upper bounds on both the execution time and the number of abortions of transactions.

Future work include the proposal of a robust variant of Polka CM, by introducing static priorities that would allow to relax the functional and timing constraints of transactions. Indeed, in the case where a transaction becomes timely incorrect, there is actually no guarantee that all timely correct transactions will progress.

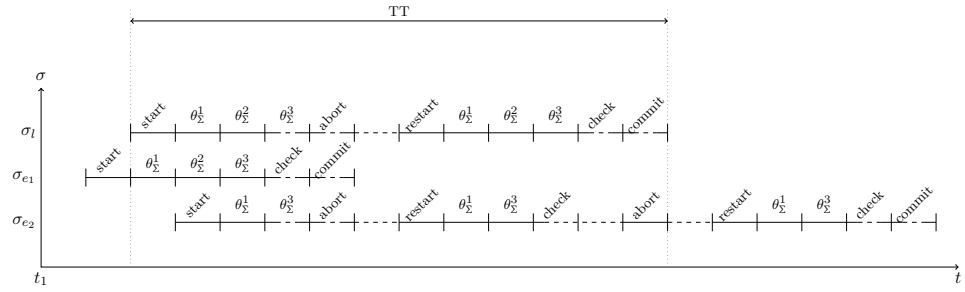

Figure 8: Illustration of the proof of Theorem 5.5 (cases ii.c and ii.d.1)

σ_l is necessarily *timely correct*: either σ_l aborts ($k_{e_1} > k_l$) or σ_l commits ($k_{e_2} \leq k_l$).

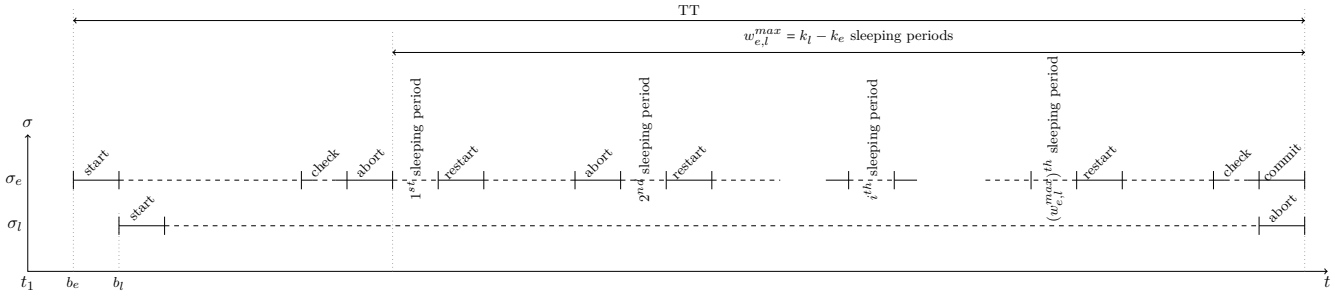

Figure 9: Illustration of the proof of Theorem 5.5 (cases ii.d.3 and ii.d.4)

σ_l is necessarily *timely correct*: σ_e will ultimately abort σ_l when reaching the maximum number of sleeping periods.

REFERENCES

[1] A Barros, P Meumeu Yomsi, and L-M Pinho. 2016. Response time analysis of hard real-time tasks sharing software transactional memory data under fully partitioned scheduling. In *Proceedings of the11th IEEE Symposium on Industrial Embedded Systems (SIES)*. IEEE.

[2] A Barros and L-M Pinho. 2014. Non-preemptive scheduling of real-time software transactional memory. In *Proceedingsof the 27th International Conference on Architecture of Computing Systems (ARCS)*. ACM.

[3] S Cotard, A Queudet, J-L Béchennec, Sébastien Faucou, and Yvon Trinquet. 2015. STM-HRT: A Robust and Wait-Free STM for Hard Real-Time Multicore Embedded Systems. *ACM Trans. Embedded Comput. Syst.* 14, 4 (2015), 1–25.

[4] D Dice, Y Lev, M Moir, D Nussbaum, and M Olszewski. 2009. Early experience with a commercial hardware transactional memory implementation. In *Proc. 14th Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*. ACM, 157–168.

[5] Son Dinh, Jing Li, Kunal Agrawal, Chris Gill, and Chenyang Lu. 2017. Blocking analysis for spin locks in real-time parallel tasks. *IEEE Transactions on Parallel and Distributed Systems* 29, 4 (2017), 789–802.

[6] M El-Shambakey and B Ravindran. 2012. STM concurrency control for embedded real-time software with tighter time bounds. In *Proceedings of Design Automation Conference (DAC)*. IEEE.

[7] M El-Shambakey and B Ravindran. 2013. On real-time STM concurrency control for embedded software with improved schedulability. In *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE.

[8] Kapali P. Eswaran, Jim N Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The notions of consistency and predicate locks in a database system. *Commun. ACM* 19, 11 (1976), 624–633.

[9] Rachid Guerraoui, Maurice Herlihy, Michal Kapalka, and Bastian Pochon. 2005. Robust contention management in software transactional memory. In *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL" 05)*.

[10] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. 2005. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. ACM, 258–264.

[11] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 175–184.

[12] Theo Haerder and Andreas Reuter. 1983. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)* 15, 4 (1983), 287–317.

[13] Tim Harris, James Larus, and Ravi Rajwar. 2010. Transactional memory. *Synthesis Lectures on Computer Architecture* 5, 1 (2010), 1–263.

[14] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-free synchronization: Double-ended queues as an example. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*. IEEE, 522–529.

[15] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. 2003. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 92–101.

[16] Ma. Herlihy and J.E.B Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 289–300.

[17] C Hung, J Yen, L En, S Diestelhorst, M Pohlack, and M Hohmuth. 2010. ASF: AMD64 Extension for Lock-Free Data Structures and Transactional Memory. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE/ACM, 39–50.

[18] Damien Imbs, José Ramón González De Mendivil Moreno, and Michel Raynal. [n. d.]. *On the consistency conditions of transactional memories*. Technical Report. Issue 1917, IRISA, University of Rennes, France.

[19] C Jacobi, T Slegel, and D Greiner. 2012. Transactional memory architecture and implementation for IBM system Z. In *Proceedings*

of the 45th Annu. IEEE/ACM Int. Symp. Microarchitecture. IEEE/ACM, 25–36.

[20] Eric H Jensen, Gary W Hagensen, and Jeffrey M Broughton. 1987. *A new approach to exclusive data access in shared memory multiprocessors*. Technical Report. UCRL-97663, Lawrence Livermore National Laboratory.

[21] Xu Jiang, Nan Guan, He Du, Weichen Liu, and Wang Yi. 2020. On the Analysis of Parallel Real-Time Tasks with Spin Locks. *IEEE Transactions on Computers (Early Access)* (2020).

[22] Lee Kangmin and Jo Heeseung. 2018. ParaTM: Transparent Embedding of Hardware Transactional Memory for Traditional Applications. *IEEE Access Journal* 6 (2018), 45417–45426.

[23] Toufik Sarni, Audrey Queudet, and Patrick Valduriez. 2009. Real-time support for software transactional memory. In *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA'09. 15th IEEE International Conference on*. IEEE, 477–485.

[24] William N Scherer III and Michael L Scott. 2005. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. ACM, 240–248.

[25] M Schoeberl, F Brandner, and J Vitek. 2010. RTTM: Real-time transactional memory. In *Proceedings of the 2010 ACM symposium on Applied Computing*. ACM, 326–333.

[26] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.