J Supercomput (2015) 71:1095–1115
DOI 10.1007/s11227-014-1356-x

A priority scheduling for TM pathologies

Chia-Jung Chen · Rong-Guey Chang

Published online: 23 December 2014

Abstract Developing a parallel program on Chip multi-processors (CMPs) is a critical and difficult issue. To overcome the synchronization obstacles of CMPs, transactional memory (TM) has been proposed as an alternative control concurrency mechanism, instead of using traditional lock synchronization. Unfortunately, TM has led to seven performance pathologies: DuelingUpgrades, FutileStall, StarvingWriter, StarvingElder, SerializedCommit, RestartConvoy, and FriendlyFire. Such pathologies degrade performance during the interaction between workload and system. Although this performance issue can be solved by hardware, the software solution remains elusive. This paper proposes a priority scheduling algorithm to remedy these performance pathologies. By contrast, the proposed approach can not only solve this issue, but also almost achieve the same performance as hardware transactional memory systems.

Keywords Performance pathologies · Transactional memory · Priority scheduling

1 Introduction

Using lock synchronization to develop a parallel program on chip multi-processors (CMPs) is difficult and error-prone. To exploit the instruction level parallelism (ILP), many solutions have been proposed to resolve challenges including interconnection network [1] and scheduling problem [2]. For example, the sorting problem executed in parallel has been addressed for interconnection network [3]. Arabnia et al. proposed

C.-J. Chen · R.-G. Chang (✉)
Advanced Institute Manufacturing with High-TECH Innovations and Department of Computer Science Information Engineering, National Chung Cheng University, Chiayi, Taiwan
e-mail: rgchang@cs.ccu.edu.tw

C.-J. Chen
e-mail: ccj98p@cs.ccu.edu.tw

Table 1 Lock-based vs. transactional memory

	Lock-based	TM
Programming friendly	Difficult and error-prone	Easy and simple
Deadlock and livelock prevention	Programmers require to adopt a locking policy	Prevented entirely
Priority inversion	Low-priority thread holding exclusive access to a resource	High-priority transactions can abort conflicting lower priority transactions

parallel sort algorithms to resolve these overheads [4,5]. Their work was efficient using pipelined-sort and MultiRing-Sort. In contract, one alternative to lock-based synchronization to control concurrency is transactional memory (TM) [6]. For lock-based synchronization, programmers must lock and unlock shared data carefully in a parallel program to avoid significant performance degradation, deadlock, and livelock. TM system can exploit the ILP of an application during execution; it provides re-use buffer called read set and write set to replace wrong operations. Unlike scheduling, TM can prevent error and avoid the addition of extra resources to improve performance. Conversely, TM prevents deadlock and livelock entirely by employing priority mechanism. High-priority transactions can abort the conflicting transactions with lower priority, and the mechanism can also prevent priority inversion. The main contrast between lock-based synchronization and TM is shown in Table 1.

There are three major manageable mechanisms in TM: version management (VM), conflict detection (CD), and conflict resolution (CR). In hardware transactional memory (HTM), read-sets and write-sets are investigated to control the concurrency mechanism. The selection to store the new data in memory or a log file is managed using a VM mechanism. To determine whether the data version in read-sets and write-sets is conflicting or not, CD is called. When a conflict occurs, we must achieve a CR to abort one of transactions, and thereby allow the program to continue its execution.

The following describes the three manageable mechanisms that exist on HTM.

- *Version management* (VM) is a method to store newly written new existing and old values. The newly written values are that when the transaction commits, they need to be able to store in the right place and be visible to other transactions. When the transaction aborts, the old values are needed to be written back from the previous transaction. There are two policies on version management: eager and lazy. Eager version management stores old values on an undo log [7–9] and the newly written values are in place. When the transaction aborts, the old values need to be written back from the undo log. Because the new values are already in place, its commits faster than aborts. But, slow aborts make transactions more contentious. On the other hand, lazy version management not only saves the old values in place, but also saves the newly written values to a write buffer. When the transaction commits, it needs to move data into the right place from the write buffer. This technique makes aborts fast.
- *Conflict detection* (CD) is the policy to examine read-sets and write-sets for detecting data conflicts. There are also two policies on conflict detection: eager and lazy.

Eager conflict detection is a hardware transactional memory that detects a conflict on individual memory references. Eager conflict detection reduces the conflict times because it uses stalls rather than aborts. Lazy conflict detection is a hardware transactional memory that detects conflicts when transactions commit. Lazy conflict detection can mitigate the impact of some conflicts and has a batch of checking conflicts [10,11].

– *Conflict resolution* (CR) is the policy to do when a conflict is detected. When we use eager conflict detection, we resolve the conflicts on each memory reference. The conflict resolution policy is stalling the requester, aborting the requester, or aborting the others. Lazy conflict detection resolve conflicts when a committer checks the conflicts between its own transactions and other transactions. The resolution policy can abort all others, stall or abort the committer.

Based on VM, CD, and CR, we can specify these design points as LL (Lazy CD/Lazy VM/Committer Wins) systems, EL (Eager CD/Lazy VM/Requester Wins) systems, and EE (Eager CD/Eager VM/Requester Stalls) systems. Please note that there is no LE (Lazy CD/Eager VM) system on HTM at present. LE system waits until commit time to detect conflicts so that the transactions become "zombies", continue executing, waste resources, and even abort.

– Lazy CD/Lazy VM/Committer Wins(LL) LL (Lazy CD/Lazy VM/Committer Wins) systems store new value in the write buffer and check if conflicts have occurred. When the transaction commits, it writes the data to the correct place from the write buffer and check whether the read-sets and write-sets conflict or not. There is some research like TCC [12] and Bulk [10] using the LL system. When transaction commits, it will acquire the commit token [11] or the commit bus [10]. This means that there is only one transaction that can commit (including writing data and checking conflict). There is a conflict when a transaction has read the memory address in the committing transaction write-set. The conflict resolution is committer wins. No matter whether the transaction starts earlier than the transaction which owns the commit token, it should be aborted. This system guarantees forward progress because it must have a transaction commit which aborts the conflicting one. That is, the committing transaction always has a higher priority to finish its own execution.
– Eager CD/Lazy VM/Requester Wins(EL) EL(Eager CD/Lazy VM/Requester Wins) systems detect conflicts on individual memory references and store newly values to the writer buffer. When the transaction commits, it needs to update the newly data from the write buffer, such as LTM [13]. When the transaction requests to access the conflict memory address, the requester has a higher priority than others. So, the one that is conflicted with the requester must abort. Because the old values remain in the right place until commit, it means that transactions can abort quickly. The EL policy appeals to early adopters because it is compatible with existing coherence protocols that always respond to coherence requests [14].
– Eager CD/Eager VM/Requester Stalls(EE) There are some existing EE (Eager CD/Eager VM/Requester Stalls) systems, such as LogTM variants [7,8,15]. This system also detects conflicts on individual memory references, and the newly date is already in place. The old values will write into a per-thread log. EE stalls

requester when conflicts and aborts only if a stall is on a potential deadlock. EE records the transaction start time for detecting potential cycles. When a transaction has been stalled by an older transaction, it could not be stalled. Eager version management lets commit become faster, but abort becomes slower because it need to restore the old values form the log. If the transaction has lots of newly data, it possibly makes the private cache overflow on Lazy VM. So the advantage on Eager VM is that we do not worry about buffer overflow problem. HMTM [6] is similar to EE systems. It stores both new and old values in the cache. HMTM, like EE, is quicker when conflict occurs. Unlike EE, HMTM aborts requesters instead of stalling them.

Bobba et al. [14] reported that TM spends much time on useless execution behaviors including abort, commit, stall, and backoff. They also indicated the performance pathologies that harm performance of TM programs and presented a hardware solution to solve these pathologies. Most previous research has accelerated programs using hardware to improve TM program performance, and only a few software transactional memory (STM) approaches were presented. Scherer and Scott [16] indicated that conflict resolution is the critical goal of STM to avoid too many pathologies to degrade performance. Although HTM can achieve high performance, modifying hardware architecture is costly [17]. In this paper, we present another way to accelerate TM programs by parsing transactions in advance with a software approach, and applying a priority scheduling scheme to resolve conflict for TM. Using software costs less than using hardware and our approach can achieve the same, or an even higher performance, by comparing our research with the work presented in [14].

The rest of this paper is organized as follows. Section 2 discusses the related work. We describe the performance pathologies of HTM and introduce their hardware solutions in Sect. 3. Section 4 presents the proposed priority scheduling approach. We then describe the implementation details and show the result in Sect. 5. Finally, we conclude this paper briefly in Sect. 6.

2 Related work

HTM systems may modify processors, cache, and bus protocol, to control transactions such as LogTM variants [7,8], LTM [13,18,19], TM coherence and consistency(TCC) [12,20], and Bulk [10]. The Log-based TM is based on the EE systems and uses MOESI directory protocol. Although it can detect conflicts and commit quickly, they must roll back to the old values when a transaction aborts. Furthermore, the LL system is similar to the work presented in Bulk [10] and (TCC) [12]. TCC stores the new values in the L1 cache and overwrites L2 cache and memory after transactions commit, and thus detects any conflicts at that time. The EL system sample is LTM [13] that allows the old value to remain in the main memory and stores the new value in the cache. LTM detects conflict for each memory reference. HTM implementations can achieve better performance with hardware complexity, but are limited to restricted semantics such as object conflict detection. In contrast, STM can resolve the issues of transactional memory semantics in a runtime library and or a programming language with some hardware support such as an atomic compare and swap operation [21]. Compared with HTM, STM is more flexible, easier to be modified and integrated with existing systems and language features, but they incur higher overhead. Thus, some researchers proposed approaches to resolve this issue with contention manager [16,22,23].

Other researchers have examined TM with a given priority. First, Justin and Daniel [24] extended a TM contention manager for the user-defined priority-based transactions. Their system focused on real systems, or some strict systems, requiring more restrictions. In contrast, our method does not have any limitation on systems. Additionally, Karma [16,23] had tracked the cumulative number of blocks opened by a transaction as its priority. When a transaction commits, it resets the priority values to zero. When the transaction opens the block, it increases their priority values. The contention manager compares the transactions's priority and aborts the smaller one when a transaction conflict occurs. When a transaction retries, it will increase the transaction priority value. In general, they are delay-based contention manager, like backoff, Karma, and Polka. The detailed performance analysis of different designs has recently been studied by Ansari et al. [25]. They suggested that priority will be the efficient solution to resolve all pathologies. Spear et al. [22] identified downsides of priority-related mechanism, and the critical challenge was read visibility. In addition to the techniques above, a compiler optimization becomes a compromise among competing imperatives. In this paper, we present an alternative approach to perform read visibility with compiler and decide when a particular transaction executes [26]. The most noteworthy difference with this manager and our approach is that their manager does not analyze transactions in advance like read visibility. By comparison, we can analyze transactions to obtain more information and apply it to modify the priority values properly. Besides, a hybrid transactional memory has been presented to implement TM in software so that it can use best-effort HTM to boost performance [12]. Thus, programmers can develop and test transactional programs on existing systems with HTM support.

To understand the software transaction memory is to use a priority to control transaction based on the size of transaction, the percentage of store instructions in each transaction, the progress of thread which contains the transaction, and the number of retries.

3 Background

The seven performance pathologies simplify the interaction of TM system design and program transactions, leading to interesting execution patterns that can affect performance. Pathologies degrade performance by preventing a transaction from making progress or performing unusable work, which is then discarded when a transaction aborts. They are described as follows.

DuelingUpgrades (DU): When two concurrent transactions read and later attempt to modify the same cache block, this pathology occurs. Because both transactions add the block to their read-sets, the conflict is detected when they write the same data, causing one of the transactions to abort. This behavior is pathologic only for EE systems because of their slower aborts. The requester-stalls resolution policy further exacerbates the problem. For example, consider the case in Fig. 1. The committing

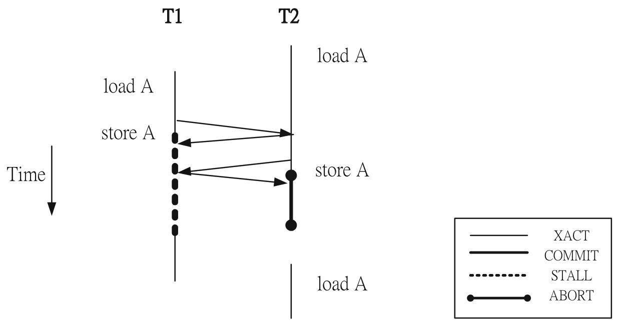
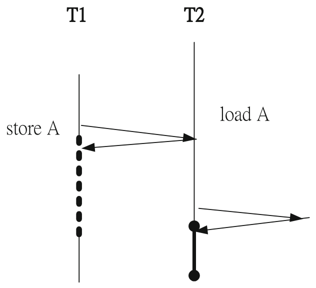


Fig. 1 DuelingUpgrades pathology

Fig. 2 FutileStall pathology



transaction may first stall on one that abort (that is, the FutileStall pathology). The two transactions begin and read the same block: transaction T1 then attempts to write the block but stalls due to the conflict. When transaction T2 also tries to write, a deadlock is detected and the system aborts the requester to resolve the possible deadlock. T2 stalls trying to read the now-exclusive block until T1 commits when it restarts. T1 can repeat the conflict when it immediately starts another identical transaction, and loses the conflict resolution because it becomes the requester of a possible deadlock.

FutileStall (FS): Eager conflict detection may cause a transaction that ultimately aborted to stall for another transaction. In this case, the stall is unnecessary, because it did not resolve a conflict with a transaction that performs useful work. Eager version management exacerbates this pathology because the HTM system must restore the old values to maintain isolation on its write-set. Thus, a transaction can stall on another transaction that ultimately aborts and continues to stall while the system restores the old values from the log. Transaction T1 is stalled to wait for transaction T2 that ultimately aborts, as shown in Fig. 2.

StarvingWriter (SW): This pathology occurs when a transactional writer conflicts with a set of concurrent transactional readers. The writer stalls to wait for the readers to finish their transactions and release isolation. The writer may starve if new readers arrive before the existing readers commit [27], as illustrated in Fig. 3. The writer is blocked by a series of committing readers. In a more favorable case, the readers continue their jobs and only the writer starves. In the least favorable case, none of the transactions make progress, because the readers encounter a cyclic dependence on the

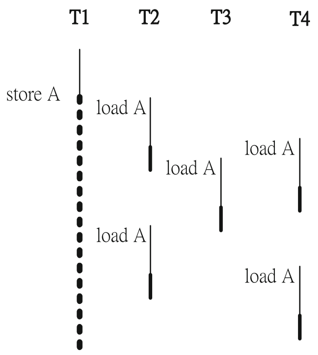


Fig. 3 StarvingWriter pathology

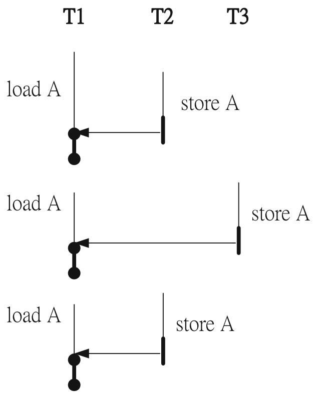


Fig. 4 StarvingElder pathology

writer after reading the block, abort (releasing isolation), but they then retry before the writer acquires access.

StarvingElder (SE): The pathology may occur in a lazy conflict detection system and a "committer-wins" policy because the system allows smaller transactions to starve longer transactions [11]. Small transactions naturally reach their commit phase faster and the committer-wins policy allows repeated small transactions to abort a transaction. The resulting load imbalance may have a broad performance repercussion. For example, the small transactions executed by threads T2 and T3 repeatedly abort transaction T1, as shown in Fig. 4.

SerializedCommit (SC): HTM systems with lazy conflict detection serialize transactions during commit to preserve a global serial order. Thus, committing transactions may stall while waiting for other transactions to commit. This type of case always happens in a program with many small transactions. However, the overhead can be reduced if the finishing transaction is guaranteed to commit by a committer-wins resolution policy [10, 11]. Figure 5 illustrates this pathology. In this figure, the commits are serialized due to the limitations on HTM systems, although none of the transactions conflict.

RestartConvoy (RC): This pathology happens in HTM systems with a lazy conflict detection. When a committing transaction conflicts with (and aborts) multiple instances of the same static transaction, the aborted transactions restart simultane-

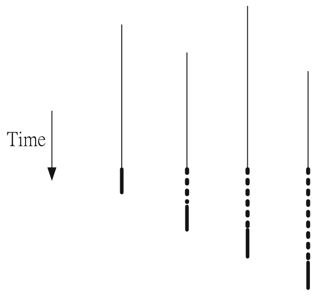


Fig. 5 SerializedCommit pathology

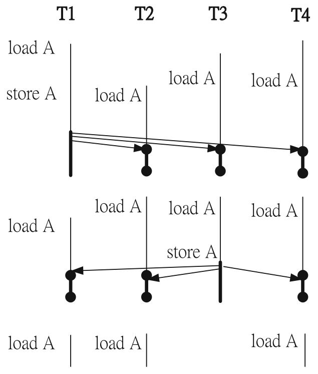
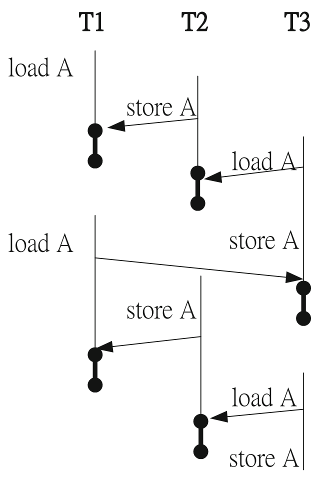


Fig. 6 RestartConvoy pathology

ously, compete for system resources, and then finish together due to their similarity. The crowd of transactions competes to commit and the winner aborts the others. Convoys can persist indefinitely if a thread that wants to commit a transaction rejoins the competition before all other transactions have a chance to commit [28]. A transaction convoy degrades performance in the following two ways. First, convoys force the program to be serialized in a single transaction when other parts of the program can be executed concurrently. Second, the transactions that are restarted increasingly contend for system resources. The convoy effect arising from restarting transactions is shown in Fig. 6. As transaction T1 commits, the transactions of other threads abort. Those threads restart and complete almost simultaneously, and one commits again and the rest of the transactions abort. As a result, the convoy may persist if threads which pass the transaction return and re-enter the convoy.

FriendlyFire (FF): Figure 7 illustrates this pathology, which occurs when one transaction conflicts with and aborts other transactions, which are then subsequently aborted before committing any useful work. In the least favorable case, this pathology repeats indefinitely with concurrent transactions, leading to abort each other and result in livelock. Because a simple requester-wins policy exhibits the FriendlyFire pathology and frequently results in livelock under high contention [10,16,29,30], the baseline EL policy of the GEMS simulator uses randomized linear backoff after an abort occurs [31]. VTM also employs eager conflict detection and lazy version management, but does not specify a conflict resolution policy [32].

Fig. 7 FriendlyFire pathology



4 The proposed priority scheduling algorithm

In this section, we present our priority scheduling algorithm to resolve the seven performance pathologies and show how to set the priority, including static priority and dynamic priority. Considering these pathologies described in Sect. 3, we discuss each pathology and compare hardware solutions with our approach.

4.1 Algorithm

The symbols used in the algorithm are defined as follows. For a transaction T,

- T_i is the ith transaction,
- $P(T)$ is the priority of transaction T,
- $P_S(T)$ is the static priority of transaction T,
- $P_D(T)$ is the dynamic priority of transaction T,
- $C(T)$ is the total clock cycles to execute transaction T,
- TC is the total clock cycles of the whole program,
- $S(T)$ is the different store address counts of transaction T,
- $\mathrm{LS}(T)$ is the number of all loads and stores address counts of transaction T,
- $\mathrm{CE}(T)$ is the current execution cycles of transaction T,
- $N_R(T)$ is the number of RETRY times of transaction T,
- $\mathrm{BO}(T)$[1] is backoff cycles of transaction T, and
- $\mathrm{BO}_{\mathrm{base}}(T)$ is the original backoff cycles of transaction T.
- $N(T)$ is the successful counts of transaction T,
- $\mathrm{CE}_S(T)$ is the current execution time (cycles) when transaction T begins.

Figure 8 shows the proposed priority scheduling algorithm. First, we must obtain the static priority of each thread of a transaction. The static priority consists of the

[1] After T aborts, it waits backoff time to restart.

```
Initial:
begin
    For a transaction $T$
        Assign $P_S(T)$ to $T$
end

Manage Data:
begin
    Update $P_D(T)$ based on $CE(T)$

    For transactions $T$ and $T'$, if a data conflict occurs
    between $T$ and $T'$
        if $P(T) > P(T')$
            Abort $T'$
            Update $N_R(T')$, $P_D(T')$ and $BO(T')$
        else
            Abort $T$
            Update $N_R(T)$, $P_D(T)$ and $BO(T)$
end
```

Fig. 8 The proposed priority scheduling algorithm

execution time of each transaction, and also the different store addresses of each transaction. If we know the execution steps of a transaction, including its memory reference and execution time in advance, we can use this information to set the priority. The information is defined before execution and does not change at run time; therefore, we call them "static priority". Next, we set the "dynamic priority" which is changed based on the current execution time and the number of retries at run time. Dynamic priority records the execution cycles of each transaction. If a transaction has been executed for a while, it should not be aborted by the other one. Consequently, the larger transaction does not waste the retry time. However, if a smaller transaction is always aborted by larger transactions, we should handle this case carefully by taking the number of retry times into account. Furthermore, since the priority values influence the backoff time, a transaction will have a smaller backoff time (that is, the transaction restarts faster) if it has higher priority. The transaction having the highest priority can abort others and thus will not be interrupted by other transactions. When a transaction is aborted, we must update the "dynamic priority" and backoff time.

4.2 Priority assignment

This section presents the priority setting in detail. Initially, each transaction is assigned a priority value, and these priority values will change during execution, despite different processors executing the same transaction due to the priority value being composed of static and dynamic priorities. Notice that transactions executed by different processors have the same static priority, but their dynamic priorities are different because they are determined by the current execution time and the number of retry times. Thus, this approach can prevent transactions executed by different processors from competing with each other. The traditional TM conflict resolutions are inflexible and do not consider all related issues. Therefore, this paper presents a new method by taking all related issues into account to improve the original TM conflict resolution. Most importantly, PS mitigates the performance pathologies as well as improves TM program performance. The following is the way to set the priority.

$$
\begin{aligned}
P_{\mathrm{S}}(T) &= \alpha \times (N(T) \times C(T))/\mathrm{TRS} + \beta \times S(T)/\mathrm{LS}(T)\\
\mathrm{TRS} &= \sum N(T_i) \times C(T_i),\ T_i \in \text{all} \quad \text{transactions}\\
P_{\mathrm{D}}(T) &= \gamma \times (\mathrm{CE}(T) - \mathrm{CE}_{\mathrm{S}}(T))/C(T) + \delta \times N_{\mathrm{R}}(T)\\
\mathrm{BO}(T) &= \epsilon \times (1/P(T)) \times \mathrm{BO}_{\mathrm{base}}.
\end{aligned}
$$

Please note that α, β, γ, δ, and ϵ are the initial weights. The static priority is defined based on the previous execution transactions of threads. We monitor the execution time of each transaction and store different address counts to determine the static priority. To verify whether the transaction is large or not, we record the execution time of each transaction. If the transaction often modifies the memory addresses, we assume that it spends more time and we assign it a higher priority. Notably, static priority must be defined before invoking our priority scheduler. We exemplify how to calculate the static priority as follows.

Example:

Cycles	Instruction
100	Begin transaction(id)
101	LOAD A
102	STORE B
103	STORE A
104	STORE B
105	STORE C
106	LOAD B
107	Commit transaction(id)

$C(T)$ is 107-100 = 7

The transaction stores A, B, C, so $S(T)$ is 3

Based on the priority scheme,

$$
\begin{aligned}
P_{\mathrm{S}}(T) &= \alpha \times (7 \times N(T))/\mathrm{TRS} + \beta \times S(T)/\mathrm{LS}(T)\\
&= \alpha \times (7 \times N(T))/\mathrm{TRS} + \beta \times 3/\mathrm{LS}(T)
\end{aligned}
$$

4.3 PS on TM pathologies

In this section, we exemplify the seven pathologies with our priority scheduling algorithm and identify the priority relationships between transactions.

Figure 9 illustrates DuelingUpgrades that have been solved by our approach. We assume $P(\mathrm{T2}) > \mathrm{P(T1)}$, because the execution time of T2 is longer than that of T1, while the original conflict resolution stalls and aborts the requester in a possible

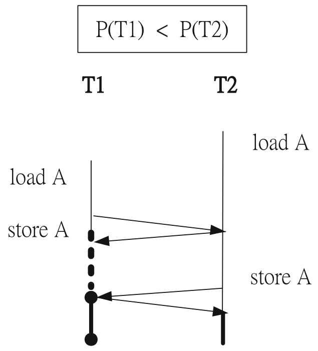


Fig. 9 The proposed priority scheduling for DuelingUpgrades pathology

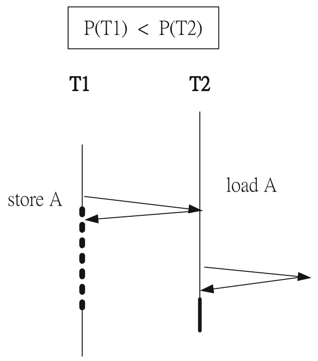


Fig. 10 The proposed priority scheduling for FutileStall pathology

deadlock. Here, we stall the requester but abort the transaction having the smaller priority. T1 makes a request to store A at the first time, thus T1 is stalled by T2 and afterwards T2 requests to store A. If we stall the requester too, it causes a deadlock. Therefore, we must abort one of the transactions. Based on the priority values, we abort the smaller one of both. A hardware solution EE_P can decrease the pathology ratio using a small write-set predictor to predict the deadlock [7]. We then anticipate that our algorithm will be able to combine with EE_P to achieve the most favorable result with the smallest pathology ratio.

FutileStall: The EE system stalls and then aborts the requester on possible deadlock. Figure 10 illustrates FutileStall as solved by our algorithm. FutileStall is similar to DuelingUpgrades, but the situation in T2 is aborted by transactions other than T1. We also assume $P(\text{T2}) > \text{P(T1)}$. The original T2 is aborted by the other transaction; but in our assumption, T2 has a higher priority and can continue executing without wasting the execution time. The hardware does not target this pathology, but can be resolved in this example. We have an advantageous position regarding this pathology.

StarvingWriter: This case also occurs in the EE system and thus it has the same conflict resolution as DuelingUpgrades and FutileStall. This case happens when readers stall writer successively and writer may starve. The hardware solution EE_{HP} extends EE_P to reduce STARVINGWRITER by allowing an older writer to abort a number of younger readers simultaneously. Our approach is similar to EE_{HP} by giving the writer higher priority than readers. Hence, the writer does not starve and waste the

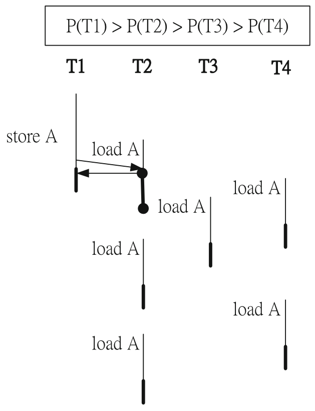


Fig. 11 The proposed priority scheduling for STARVINGWRITER pathology

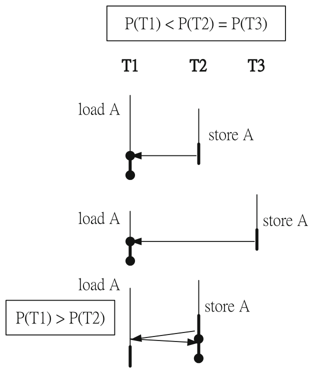


Fig. 12 The proposed priority scheduling for StarvingElder pathology

stall time. Figure 11 illustrates StarvingWriter to be solved by PS by considering the store address times to set a priority value. In these pathologies, our approach may have the same effect as the hardware solution, because the hardware solution lets the older writer abort the younger readers immediately. We also assign the writer higher priority to remedy this pathology.

StarvingElder: This problem happens in the LL system. If small transactions appear many times, as well as the conflict resolution and the committer wins, it prevents the larger transaction from having a chance to commit. As a result, the larger transaction may starve. Figure 12 illustrates StarvingElder to be overcome by the proposed priority scheduling. Our approach gives $P(\mathrm{T2}) = \mathrm{P(T3)} > \mathrm{P(T1)}$ at the initial time. To avoid low-priority transaction starves, we also considered the number of retries. When T1 restarts the transaction, our approach immediately raises the dynamic priority, and then T1 retries more times and obtains a higher priority. In Fig. 12, T1 may retry three times and be assigned the highest priority, causing $P(\mathrm{T1}) > \mathrm{P(T2)} = \mathrm{P(T3)}$. When T1 can abort other transactions, does not starve because we take retries into account when calculating dynamic priority. The hardware solution LL_B addresses RestartConvoy and also can mitigate StarvingElder and SerializedCommit. Like the LL system, LL_B is based on the committerwins policy. However, restarting transactions use randomized

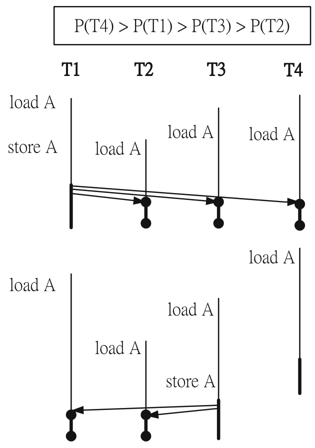


Fig. 13 The proposed priority scheduling for RestartConvoy pathology

linear backoff to delay the restart of an aborted transaction. By staggering the restart of each transaction in the group of transactions aborted by a given commit, LL_B mitigates convoy formation. But in our speculation, we think that the randomized linear backoff cannot ensure that the same case will not happen again. Thus, we assign the backoff time based on priority value. If the priority value is large, it means that the transaction should have a higher priority to be executed. Thus, we give the high-priority transaction smaller backoff time to restart immediately. In Fig. 12, it is different in the backoff time whenever the transaction restarts. SerializedCommit: This problem arises from the hardware restriction with only one write buffer. Hardware solution uses backoff time to stagger the transactions. Similarly, we also use backoff to stagger the commit time of a transaction based on priority. The high-priority transaction should execute first, and we then improve the hardware solution to achieve a higher performance. However, we do not show it because there is no fixed solution to execute the previous example. We can only ensure that using backoff time, it does not allow transactions to simultaneously commit.

RestartConvoy: Figure 13 illustrates that RestartConvoy is solved by PS. This problem affects resource contention and serialized execution. If transactions restart simultaneously, it will cause many more transactions to restart simultaneously. This result leads to these transactions causing the same problem again. The solution sets the backoff time to stagger the restart time. Although the hardware solution LL_B also sets the backoff time, ours uses priority value to decide the backoff time. If the transaction has a higher priority, we assign it a shorter backoff time and let it restart immediately. Higher priority transactions mean that they must finish earlier than others and thus we use priority value to decide backoff time. In other words, the higher priority transactions can commit earlier. Conversely, the hardware solution cannot ensure that the same case will not happen again.

FriendlyFire: The EL system detects conflicts for each memory reference and writes the new value to the writer buffer. This conflict resolution is requester wins and exponential backoff on abort. FriendlyFire causes a livelock on TM programs. The hardware solution EL_T is similar to EL, but instead of always aborting in favor of the requester,

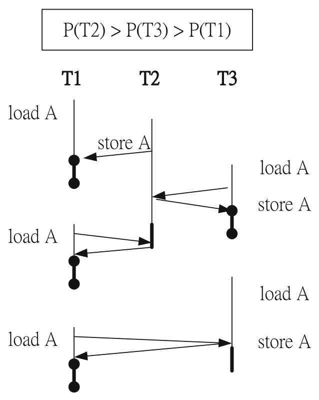


Fig. 14 The proposed priority scheduling for FriendlyFire pathology

transaction conflicts are resolved according to the logical age of the transaction, as has been completed before for implicit transactions [29] and eager alternative [10]. Our approach is similar to the EL$_T$ system, as shown in Fig. 14. Our approach can solve the problem in advance because the high-priority transactions such as T2 can abort others no matter if T2 commits or not. In this figure, the setting of which transaction has the highest transaction is not important, and therefore it then can avoid livelock. The above shows the implementation details of the proposed algorithm and the solution for pathologies. In next section, we show the experimental results regarding the occurrence ratio with the original TM setup, the hardware solution, and PS. We discovered that PS can eliminate the pathologies and improve TM program performance.

5 Evaluation

We first introduce the simulation setup and the experimental flow, and then present the evaluation results by comparing the HW solution with the proposed priority scheduling approach.

5.1 Setup

We simulate a full-system infrastructure using Simics [33] and a customized memory model built with the Wisconsin GEMS toolset [31]. The GEMS toolset leverages an existing full-system functional simulation infrastructure [31] and Simics [33] is the basis to build a set of timing simulator modules for memory system and microprocessor. Simics accurately models the SPARC architecture and GEMS supports a TM memory model. The HTM interface is implemented using “magic” instructions, which are special no-ops caught by Simics and passed to the memory model. The software components of TM systems are implemented using hand-coded assembly routines and C functions.

The setup is shown in Table 2. We model a 32-core CMP system, which is an in-order and single-issue core. Each core has 32KB private write back L1 instruction and data caches. All cores share a multi-banked 8-MB L2 cache consisting of 32 banks

Table 2 System setup

	Description
Process core	75 MHz in-order single-issue
L1 Cache	32 KB 4-way, 64-byte blocks, writeback, 2-cycle latency
L2 Cache	8 MB 8-way unified, 64-byte blocks, writeback, 15 cycle latency
Memory	4 GB, 500-cycle latency
L2 Directory	Bitvector of sharers, 6-cycle latency
Interconnect	Tiled, 64-byte links, 3-cycle link latency

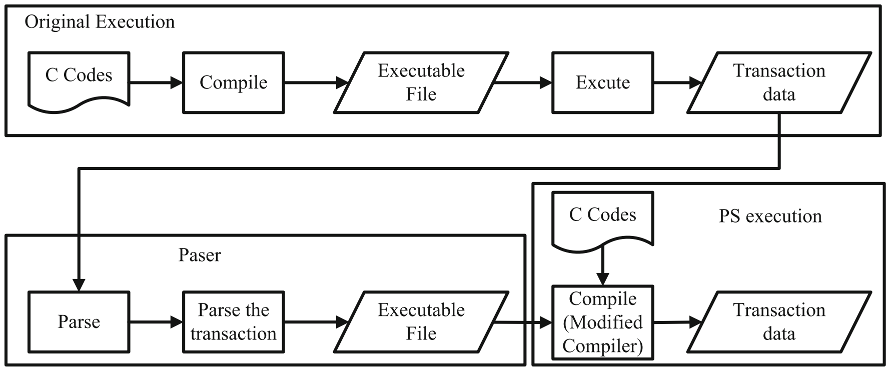


Fig. 15 Work flow

interleaved by a block address. On-chip cache coherence is maintained via an on-chip directory, which maintains a bit vector of shares and implements the MESI protocol.

Figure 15 is the work flow divided into three stages. At first, an application is compiled like the original TM execution phase. After compilation, the application is executed to obtain the transaction information. Based on this information, we can know the execution state and cycles of a transaction and then calculate the static priority of a transaction with the aid of weights. Finally, the executable file is annotated with static priority information and executed on the modified simulator, which is integrated with the proposed priority scheduler. Following these steps, we can evaluate the effect of the proposed approach on performance pathologies in Sect. 5.2.

Barnes, Cholesky, and Radiosity selected from the SPLASH [34] combining with two microbenchmarks, Btree and Deque, were performed to evaluate results. For Barnes, Cholesky, and Radiosity, these scientific programs were selected from the SPLASH benchmark suite because they can demonstrate significant critical-section-based synchronization. We replaced the critical sections with transactions while retaining barriers and other synchronization mechanisms. Btree performed a lookup or an insert and Deque enqueues/dequeues a value on the left/right of a global deque for each transaction to perform a local job and increment the global counter.

Table 3 Workload description

Benchmarks	Input	Description	Units
Cholesky	tk14.O	Factorization	1
Barnes	–	Barnes-Hut method	512
Radiosity	–	Radiosity method	–
Btree	Uniform random	BTree operation	100 K
Deque	Uniform random	Deque operation	100 K

Table 4 The percentage of execution time for EE system

	Base			HW			PS		
	DU	FS	SW	DU	FS	SW	DU	FS	SW
Deque	5.1	0.2	3.0	<0.1	<0.1	<0.1	0.3	0.3	0.5
Cholesky	0.9	<0.1	0.4	<0.1	<0.1	<0.1	<0.1	<0.1	0.2
Btree	1.1	<0.1	0.8	0.9	12.1	0.3	0.2	2.8	0.2
Barnes	0.2	<0.1	0.6	<0.1	<0.1	<0.1	<0.1	0.2	0.4
Radiosity	1.1	<0.1	0.5	<0.1	<0.1	<0.1	0.1	<0.1	0.2

Table 5 The percentage of execution time for EL system

	Base	HW	PS
	FF	FF	FF
Deque	3.2	<0.1	<0.1
Cholesky	<0.1	<0.1	NA
Btree	0.8	<0.1	1.1
Barnes	<0.1	<0.1	<0.1
Radiosity	<0.1	<0.1	<0.1

Table 6 The percentage of execution time for LL system

	Base			HW			PS		
	SE	SC	RC	SE	SC	RC	SE	SC	RC
Deque	<0.1	<0.1	<0.1	0.1	0.6	<0.1	0.2	<0.1	<0.1
Cholesky	<0.1	<0.1	0.6	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
Btree	0.2	2.3	<0.1	0.2	2.1	<0.1	<0.1	<0.1	<0.1
Barnes	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1	<0.1
Radiosity	<0.1	<0.1	<0.1	NA	NA	NA	<0.1	<0.1	<0.1

5.2 Results

In this section, we present the results of pathologies ratio and the performance for the workloads previously introduced. Tables 3, 4, 5, 6, and 7 show the comparison between the baseline result and three methods for seven pathologies. We discovered

Table 7 The percentage of execution time for PS with write-set predictor

	PS + Write-set predictor		
	DU	FS	SW
Deque	<0.1	<0.1	<0.1
Btree	0.7	9.1	<0.1
Cholesky	<0.1	<0.1	<0.1

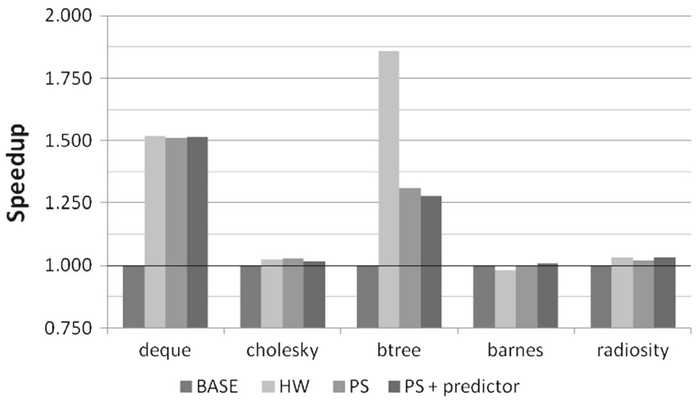


Fig. 16 Speedup for EE system

that the hardware solution can eliminate these pathologies. In contrast, PS mitigates most of these pathologies when compared with the baseline. FutileStall occurs in Btree because there were numerous reads in it. Although the pathology ratio increased, the speedup still achieved a speed of 1.5. Although sometimes PS cannot achieve the highest performance, PS + predictor can achieve the highest performance because the write-set predictor can reduce the percentage of wrong cases, as shown in Table 7. Therefore, PS is an efficient approach to resolve seven pathologies. The read visibility of the proposed work is a better way to improve the performance of different STMs. In summary, PS is a comprehensive solution and can achieve better performance.

Figure 16 shows the speedup of the EE system. The average speedups of HW, PS, and PS + predictor were 1.282, 1.173, and 1.171. Although PS can achieve a good result, PS + predictor can outperform it with the help of prediction and achieve the best maximum speedups in the EE system.

Figure 17 only shows the speedups of HW and PS for the LL system because the write-set predictor was unnecessary for this case. From the result, HW and PS only reduce the occurrences of performance pathologies a little, thus resulting in the speedup of 1.017 and 1.015.

In Fig. 18, the average speedups of HW and PS for EL system were 1.569 and 1.541, respectively. Notice that the write-set predictor was also needed to perform for this case and the only pathology happening in the EL system was friendly fire. After analyzing the result, PS can overcome friendly fire quickly to make the performance better.

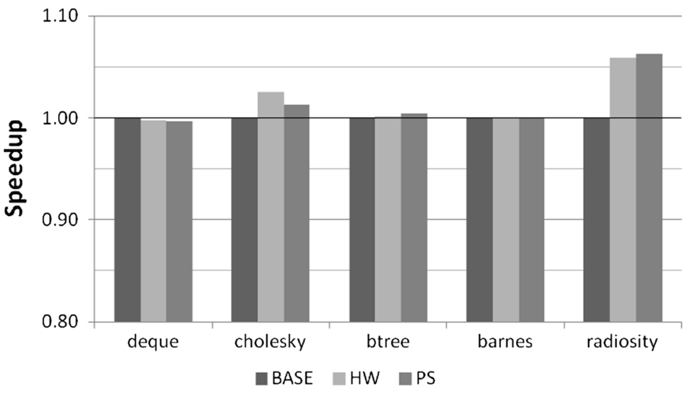


Fig. 17 Speedup for LL system

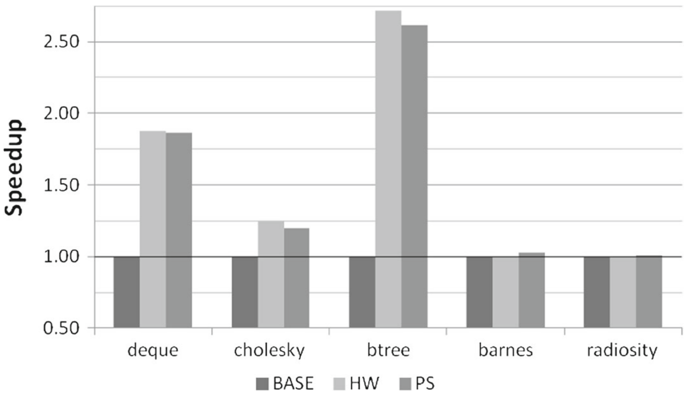


Fig. 18 Speedup for EL system

6 Conclusion

To improve the performance of transactional memory, we present a novel scheduling approach to resolve conflicts of seven performance pathologies in this paper. The proposed approach is an alternative to the accelerate TM program via parsing transactions in advance, and then rapidly assign a thread a priority to enable a positive schedule. When compared with HTM, our approach can almost achieve the same performance as HTM by adjusting static priority and dynamic priority to remedy both these pathologies without a higher hardware cost, with the exception of the EE system. However, even for the EE system, the performance of our approach is equal to that of the HTM with a predictor.

References

1. Arabnia HR, Oliver MA (1989) A transputer network for fast operations on digitised images. In: Computer graphics forum, vol 8. Wiley Online Library, pp 3–11

2. Lakshmanan K, Kato S, Rajkumar R (2010) Scheduling parallel real-time tasks on multi-core processors. In: IEEE, 2010 IEEE 31st, Real-Time Systems Symposium (RTSS), pp 259–268
3. Arabnia HR (1990) A parallel algorithm for the arbitrary rotation of digitized images using process-and-data-decomposition approach. J Parallel Distrib Comput 10:188–192
4. Arabnia HR, Smith JW (1993) A reconfigurable interconnection network for imaging operations and its implementation using a multi-stage switching box. In: Proceedings of the 7th annual international high performance computing conference, pp 349–357
5. Bhandarkar SM, Arabnia HR (1995) The refine multiprocessorxtheoretical properties and algorithms. Parallel Comput 21:1783–1805
6. Herlihy M, Moss JEB (1993) Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, New York, NY, USA, pp 289–300
7. Moore KE, Bobba J, Moravan MJ, Hill MD, Wood DA (2006) Logtm: Log-based transactional memory. In: Proceedings of the 12th IEEE Symposium on High-Performance Computer Architecture, IEEE Computer Society, Austin, Texas, USA, 2006, pp 258–269
8. Yen L, Bobba J, Marty MR, Moore KE, Volos H, Hill MD, Swift MM, Wood DA (2007) Logtm-se: Decoupling hardware transactional memory from caches. In: Proceedings of the 13th IEEE Symposium on High-Performance Computer Architecture., HPCA'07Scottsdale, AZ, pp 261–272
9. Yan Z, Jiang H, Tan Y, Feng D (2013) An integrated pseudo-associativity and relaxed-order approach to hardware transactional memory. ACM Trans Architect Code Optim TACO 9(4):42
10. Ceze L, Tuck J, Cascaval C, Torrellas J (2006) Bulk disambiguation of speculative threads in multiprocessors. In: Proceedings of the 33rd Annual International Symposium on Computer Architecture, ISCA'06, Washington, DC, USA
11. Hammond L, Wong V, Chen M, Carlstrom BD, Davis JD, Hertzberg B, Prabhu MK, Wijaya H, Kozyrakis C, Olukotun K (2004) Transactional memory coherence and consistency. In: Proceedings of the 31st Annual International Symposium on Computer Architecture, IEEE Computer Society, Washington, DC, USA
12. Damron P, Fedorova A, Lev Y, Luchango V, Moir M, Nussbaum D (2006) Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, (PPoPP'06), New York, USA
13. Ananian CS, Asanovic K, Kuszmaul BC, Leiserson CE, Lie S (2005) Unbounded transactional memory. In: Proceedings of the 11th International Symposium on High-Performance Computer Architecture, San Francisco, CA, USA, pp 316–327
14. Bobba J, Moore KE, Volos H, Yen L, Hill MD, Swiftand MM, Wood DA (2007) Performance pathologies in hardware transactional memory. In: Proceedings of the 34th annual international symposium on Computer architecture, New York, NY, USA, pp 387–394
15. Moravan MJ, Bobba J, Moore KE, Yen L, Hill MD, Liblit B, Swift MM, Wood DA (2006) Supporting nested transactional memory in logtm. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, USA, pp 359–370
16. Scherer WN III, Scott ML (2005) Advanced contention management for dynamic software transactional memory. In: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing., PODC'05NU, USA, New York, pp 240–248
17. Shriraman A, Dwarkadas S, Scott ML (2010) Implementation tradeoffs in the design of flexible transactional memory support. J Parallel Distrib Comput 70(10):1068–1084
18. Quislant R, Gutierrez E, Plata O, Zapata EL (2013) Ls-sig: Locality-sensitive signatures for transactional memory. IEEE Trans Comput 62(2):322–355
19. Bobba J, Goyal N, Hill MD, Swift MM, Wood DA (2008) Tokentm: Efficient execution of large transactions with hardware transactional memory. ACM SIGARCH Comput Archit News 36(3):127–138
20. Dalessandro L, Carouge F, White S, Lev Y, Moir M, Scott ML, Spear MF (2011) Hybrid norec: a case study in the effectiveness of best effort hardware transactional memory. ACM SIGPLAN Not 46(3):39–52
21. Carlstrom BD, McDonald A, Chafi H, Chung JW, Minh CC, Kozyrakis C, Olukotun K (2006) The $atomo\sigma$ transactional programming language. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation

22. Spear MF, Dalessandro L, Marathe VJ, Scott ML (2009) A comprehensive contention management strategy for software transactional memory. In: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming, PPoPP'09, ACM, New York, NY, USA, pp 141–150
23. Scherer WN III, Scott ML (2005) Randomization in stm contention management. In: Proceedings of the 24th ACM Symposium on Principles of Distributed Computing, PODC'05, Las Vegas, NV, July
24. Gottschlich J, Connors DA (2008) Extending contention managers for user-defined priority-based transactions. In: Proceedings of the 2008 Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods
25. Ansari M, Kotselidis C, Luján M, Kirkham C, Watson I (2009) On the performance of contention managers for complex transactional memory benchmarks. In: Eighth International Symposium on IEEE Parallel and Distributed Computing, 2009. ISPDC'09, pp 83–90
26. Ansari M, Luján M, Kotselidis C, Jarvis K, Kirkham C, Watson I (2009) Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In: High Performance Embedded Architectures and Compilers. Springer, pp 4–18
27. Courtois PJ, Heymans F, Parnas DL (1971) Concurrent control with readers and writers. Commun ACM 14(10):667–668
28. Blasgen M, Gray J, Mitoma M, Price T (1979) The convoy phenomenon. In: ACM SIGOPS Operating Systems Review, New York, USA, pp 20–25
29. Rajwar R, Goodman JR (2002) Transactional lock-free execution of lock-based programs. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems ACM, New York, USA, pp 5–17
30. Fraser K, Harris T (2007) Concurrent programming without locks. ACM Trans Comput Syst TOCS 25(2):5
31. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Alameldeen AR, Xu M, Moore KE, Hill MD, Wood DA (2005) MultifacetÕs general execution-driven multiprocessor simulator(gems) toolset. Comput Archit News 33(4):92–99
32. Rajwar R, Herlihy M, Lai K (2005) Virtualizing transactional memory. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture, ISCA'05, Madison, Wisconsin, USA
33. Magnusson PS, Magnus C, Jesper E, Daniel F, Gustav H, Johan H, Fredrik L, Andreas M, Bengt W (2002) Simics: a full system simulation platform. IEEE Comput 35(2):50–58
34. Woo SC, Ohara M, Torrie E, Singh JP, Gupta A (1995) The splash-2 programs: Characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture., ISCA'95NY, USA, New York, pp 24–36