Noname manuscript No.
(will be inserted by the editor)

Leveraging Irrevocability to Deal With Signature Saturation in Hardware Transactional Memory

Ricardo Quislant · Eladio Gutierrez · Emilio L. Zapata · Oscar Plata

Received: date / Accepted: date

Abstract In hardware transactional memory (HTM), signatures have been proposed to keep track of memory locations accessed in a transaction to help conflict detection. Generally, signatures are implemented as Bloom filters that suffer from aliasing, that is, they can give rise to false conflicts. Such conflicts are more likely as signature fills (saturation), and they can lead a parallel application to perform worse than its serial version.

Irrevocability is analyzed to address the signature saturation problem. When a transaction reaches a saturation threshold, the transaction enters an irrevocable state that prevents it from being aborted. Hence, such a transaction keeps running while the others are either stalled or allowed to run concurrently. We propose an analytical model that shows this is a good solution to overcome a high contention scenario. Also, experimental evaluation shows the benefits in performance and power consumption of the proposed irrevocability mechanisms. Different saturation metrics are considered and a fixed threshold is found that yields maximum performance for the benchmarks evaluated.

Keywords Hardware Transactional Memory · Signatures · Bloom filters · Irrevocability

1 Introduction

Transactional Memory (TM) [16, 20] is an optimistic synchronization mechanism for programming shared memory multiprocessors. TM provides the user with the transaction abstraction, a construct that enforces atomicity and isolation of the sequence of memory operations that comprises the transaction. Transactions are allowed to run concurrently unless a conflict (two transactions access the same memory location and, at least, one of the accesses is a

Dept. of Computer Architecture
University of Malaga, 29071 Malaga, Spain
E-mail: {quislant,eladio,zapata,oplata}@uma.es

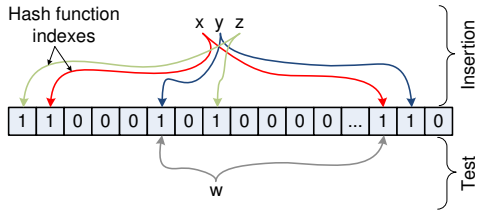

Fig. 1 Bloom filter structure, operations and aliasing.

write) is detected by the underlying transactional system, which is in charge of resolving the conflict.

Hardware manufacturers have recently released multicore/manycore processors with hardware transactional memory (HTM) extensions [14,18,37,43]. However, such extensions provide a restricted HTM system where transactions are unable to survive neither to certain operating system events, like migration, paging and scheduling, nor to transactional hardware overflow and cache evictions, amongst others. Many proposals can be found in the literature [8,29, 41] dealing with virtualization of HTM to hide those limitations to programmers. They propose different ways of virtualizing the two main mechanisms of a transactional system: conflict detection and version management.

As far as conflict detection is concerned, it is sought to detach transactional state (essentially, read and write transactional sets — RS and WS) from caches and store it in a separate and concise hardware structure that can be easily saved or moved by the operating system. Following such premises, signatures [10,41] were proposed and adopted as an effective solution to support conflict detection virtualization. Signatures, implemented as Bloom filters [5], provide a fast and concise way of dealing with large sets of addresses in fixed-size hardware but at the cost of aliasing, that is, different addresses are equally represented in the structure. Fig. 1 shows the structure of a Bloom filter comprising a bit array, initially set to zero, and a set of hash functions, which map an address to an array index. Inserting an address in the filter consists in setting to one those bits of the array pointed to by the hash functions. A test for membership is positive if the address is mapped to bits all set to one. This positive can be false if the address was not inserted and it is mapped to bits of other inserted addresses, such as w in the figure.

1.1 Motivation

Signature aliasing may cause false positives that can give rise to false conflicts between transactions, thus causing the TM system to waste time and power in resolving such conflicts. As signature fills, the false conflict rate can increase to such an extent that the parallel application can perform worse than its sequential version. We refer to this situation as *signature saturation*.

Fig. 2 shows the speedup and false positive contour lines when signature size (RS+WS) varies from 256 to 2K bits, and transaction size varies from 20

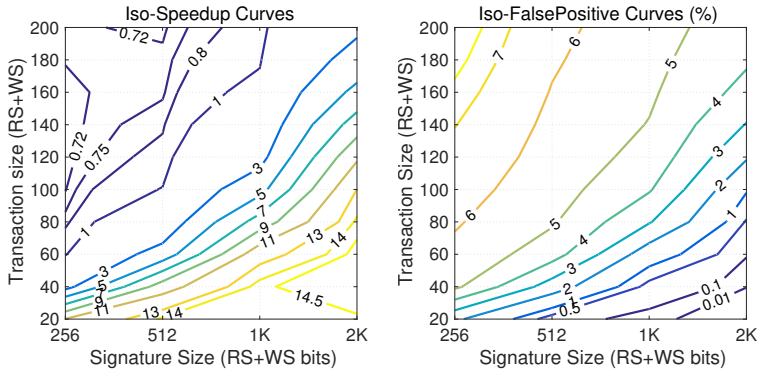

Fig. 2 Signature saturation using EigenBench. Left: Speedup contours depending on transaction size and signature size (RS: Read Set, WS: Write Set). Right: False positive contours.

to 200 cache blocks, counting both RS and WS. The values are obtained with the microbenchmark EigenBench [17] devised for orthogonal TM characteristic exploration. With this benchmark we can define transactions of any size and any amount of contention. In this case, contention is set to zero so that all conflicts among transactions are caused by false conflicts due to signature filling. We can see in the figure how large transactions or small signatures lead to scenarios of signature saturation where performance is near to sequential or worse. The percentage of false positives does not go further than 7%, but it is enough for the speedup to plummet. Our results show the similar problem with other benchmarks with large transactions such as Bayes, Labyrinth and Yada from the STAMP benchmark suite [24] (see Section 5).

1.2 Contributions

This paper analyzes irrevocability as a technique to address the signature saturation problem. Irrevocability is an execution mode of a transaction in which it cannot be aborted. This mechanism has been used for different applications [6, 36, 37, 40]. The main one is to enable the execution of irreversible operations, such as, interactive I/O and system calls within transactions. But it can be also useful for contention management and for ensuring forward progress. We, in contrast, leverage irrevocability to avoid that a parallel application underperforms the sequential one in a situation of signature saturation.

The main contributions of this paper are:

– The proposal of a solution to manage signature saturation in HTM that consists in switching to an irrevocable mode whenever a transaction reaches a given level of signature occupation. Irrevocability is requested in anticipation of a high probable scenario of false contention due to signature filling. The idea is that a saturating transaction commits as soon as possible in order to avoid aborts that could lead to very poor performance.

- The analysis of two irrevocability mechanisms, proposing a distributed implementation in the context of a tiled CMP architecture. One mechanism involves that once a transaction becomes irrevocable, all other transactions in the system stall their execution until the irrevocable transaction commits. The other mechanism allows concurrent execution of revocable transactions with the irrevocable one. Irrevocability is a feature that is being included in some commercial processors [37].
- An analytical model for the execution time of a TM system with irrevocability. The model shows how conflicts and subsequent transaction retries can harm overall performance and how irrevocability is able to avoid performance plummeting.
- An experimental evaluation of our proposals using the STAMP benchmark suite and the GEMS simulator. Both irrevocability mechanisms and the baseline TM are compared in terms of performance and network power consumption and traffic. Although one allows for more concurrency the other reduces the network traffic and power requirements. Different saturation metrics and signature designs are analyzed, finding suitable a single saturation threshold for triggering irrevocability that maximizes performance when signatures saturate. We find that performance drops severely when signatures yield more than 2.4% of false positives. We keep the benchmarks from underperforming their sequential versions and we obtain up to twice the speedup over the base system for certain configurations.

The remainder of the paper is structured as follows. Next section discusses the background and related work. Section 3 describes the irrevocability mechanisms with and without exclusion. An example of operation is shown, along with implementation details and software alternatives. An approximate analytical model for a TM system with irrevocability is performed in Section 4. Section 5 shows a comprehensive experimental evaluation. Finally, Section 6 draws the conclusions.

2 Background and Related Work

The use of hardware signatures implemented as per-thread Bloom filters was first proposed by Ceze et al. [10] to support operations that efficiently process a set of memory addresses in the context of thread level speculation and TM. Bloom filters [5] are time and space-efficient hashing structures that allow an unbounded number of element insertions with the inconvenience of aliasing, that is, a set of different elements are equally represented in the filter. Aliasing causes false positives on membership queries, i.e. the filter says that we inserted an element that we did not. Elements can be added to the filter but they cannot be deleted so the filter is false negative free.

With those characteristics, signatures were subsequently adopted by different hardware TM systems, like LogTM-SE [41] or FlexTM [34], to detach conflict detection from caches. Signatures accelerate conflict detection and provide a means to virtualize the transactional system, since having the set of

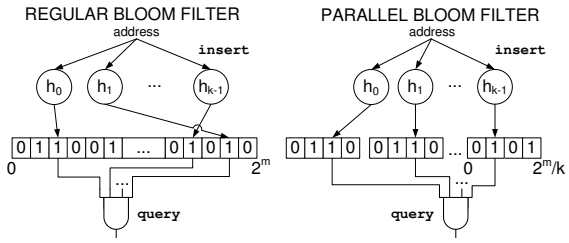

Fig. 3 Regular versus parallel Bloom filter structures.

memory addresses accessed by a transaction summarized in a single-purpose structure eases the task of migrating a thread, changing context and surviving cache victimization. Usually, signature-based TM systems have one Bloom filter per transactional data set: read set (RS) and write set (WS).

Many works have been published aimed at improving the performance of Bloom-filter-based signatures. Sanchez et al. [32] propose a parallel Bloom filter design that optimizes hardware area while maintaining the same false positive rate as a regular Bloom filter (Fig. 3 compares the structures of both regular and parallel filters). They also propose the use of the $H3$ class of hash functions [9]. Yen et al. [42] propose Page-Block-XOR hashing (PBX) to further reduce hardware area for signatures. In addition, several works deal with reducing false positive rate to enhance performance of large transactional codes [11, 27, 28, 42]. However, although these proposals can manage larger transactions than conventional signatures, signature saturation occurs as well.

Regarding irrevocability in the context of HTM, two main applications can be identified: supporting operations that cannot be rolled back within transactions (like I/O and system calls), and ensuring transaction forward progress when contention or hardware overflow occur. To provide these practical applications and to make their implementation simple, Blundell et al. [6] propose OneTM, a TM system that allows one overflowed transaction at a time. Two versions of the system are proposed, OneTM-Serialized and OneTM-Concurrent. The first of them stalls all other threads running transactional and non-transactional code whenever one transaction overflows. On the other hand, OneTM-Concurrent allows other code to execute concurrently with a single overflowed transaction by maintaining per-memory-block persistent transactional metadata, both in caches and main memory. Section 3.5 performs a qualitative comparison between OneTM and our irrevocability proposals.

To ensure transaction forward progress in the face of contention and hardware overflow, different solutions were proposed. Some best-effort systems, such as Intel Haswell [38], do not provide any special progress guarantees in hardware. To ensure progress, a software fallback path must be provided by the programmer. Other proposals [3, 25] modify the conflict resolution method in such a way that older transactions are favored over younger ones when they conflict. In this way, when a transaction becomes the oldest in the system, it will never be aborted due to contention (conflicts). This is similar to an irre-

vocable transaction, though it could be aborted and rolled back many times before being the oldest one. Systems like Vega Azul [12] provide in hardware non-speculative re-executions. When a transaction is aborted once, it rolls back and executes again non-speculatively. Finally, IBM Blue Gene/Q [37] extends this support as follows. When a transaction aborts, a runtime re-executes the transaction several times until a threshold is reached. After that, the transaction is restarted one last time in irrevocable mode.

We can find proposals of concurrency level adjustment in the field of software TM (STM), where more complex algorithms can be implemented than in HTM systems. Di Sanzo et al.[33] propose self-regulating the degree of concurrency by creating an analytical model of a specific application/platform that is subsequently used to change the number of concurrent threads in runtime. In the same research line, Rughetti et al. [31] introduce a technique combining analytical models to predict concurrency level and machine-learning algorithms to improve the accuracy of such models. Ansari et al. [4] propose a set of TCR-sampling-based algorithms to obtain a given transaction commit rate (TCR), increasing or decreasing the concurrency level depending on the sampled TCR. Finally, Didona et al. [13] explore the adaptive adjustment of concurrency level proving good for both shared-memory and distributed STM systems.

3 Irrevocability Mechanisms

In this section we describe two irrevocability mechanisms similar to those included in OneTM [6]. We propose *irrevocability with exclusion* which is similar to OneTM-Serialized. However, in contrast to OneTM, non-transactional code is allowed to run in parallel with the irrevocable transaction, since the signature of such transaction still keeps track of memory accesses. Thus, strong isolation [7] can be enforced, unlike OneTM-Serialized that needs to stall transactional and non-transactional code because the overflowed transaction cannot track its transactional metadata anymore. We also describe a mechanism of *irrevocability with no exclusion*, or just *irrevocability*, which is similar to OneTM-Concurrent. However our technique does not require extra hardware (unlike OneTM, that needs persistent transactional state in main memory) since signatures track the accesses of the irrevocable transaction. Section 3.5 gives more insight into the differences between our irrevocability mechanisms and OneTM.

3.1 Irrevocability with no exclusion

To specify the proposed irrevocability mechanisms we define two irrevocability states, one for the thread (TS) and the other for the transaction being executed by the thread (XS). Table 1 summarizes the meaning of different combinations of the states: TS stands for *Thread State* and XS is the *Transaction State*. Both

Table 1 Irrevocability states

TS	XS	Description
N	N	Normal mode
N	I	Not possible
I	N	Irrevocability triggered by other thread; Transaction runs normal, but conflict resolution favors (I,I) transaction
I	I	Transaction runs as irrevocable (exclusive state)

states can be set to either N (*Normal*) or I (*Irrevocable*). The thread that is in the irrevocable mode can set the TS state for all other threads.

Table 2 shows the protocol state machine of the distributed mechanism of irrevocability with no exclusion. Each core's L1 controller executes an instance of the state machine. The states are of the form (0/1, TS, XS). The state (0, TS, N) points out that the thread is in state TS and is executing non-transactional code. The state (1, TS, XS), on the other hand, means that the thread is in state TS and is executing a transaction in state XS. The transient state XS=NI corresponds to a normal transaction that reached the signature saturation condition, requesting and waiting to become irrevocable before transitioning to state XS=I. The top table in Table 2 shows the behaviour of the irrevocability mechanism due to thread/core events, specifically, the begin and commit of a transaction and the signature saturation condition (i.e. reach the occupation threshold). Each table cell shows the corresponding action and the resulting state after such action. The bottom table in Table 2 shows the mechanism behaviour for network events, triggered by other concurrent threads. The first three columns represent a conflict with other transaction in some state XS (N, I or NI), the next column corresponds to the granting of irrevocability, and the last two columns correspond to the broadcast of the command to change TS state (N or I) from the irrevocable transaction to the other threads.

A transaction in normal state that reaches saturation of its signature requests irrevocability, and immediately stalls (transient state XS=NI). An arbiter of the irrevocability state grants such condition to only one transaction at a time. So, if several transactions saturate their signatures simultaneously, only one of them can pass to XS=I state, whereas the others stall (XS=NI) until the former commits.[1] The transaction that becomes irrevocable notifies the rest of the threads (broadcast the command TS=I) so that they set their state to TS=I. The irrevocable transaction unstalls and continues its execution in such a state. When the irrevocable transaction commits, it broadcasts the command TS=N to the rest of the threads for them to change their state to normal. The transactions that were stalled waiting for irrevocability (state TS=NI) also get such a message, thus requesting irrevocability again.

The conflict resolution between transactions is carried out normally unless a transaction is in XS=I state. However, when a transaction conflicts with

[1] In this case, the queues of the network interconnect routers arbitrate which irrevocability request packet is enqueued first, whose owner will be granted irrevocability in the form of a token (see Section 3.4).

Table 2 Protocol for irrevocability with no exclusion

States	Thread/Core Events		
	Transaction Begin	Transaction Commit	Signature Saturation
(0,N,N)	*No Action* /(1,N,N)		
(0,I,N)	*No Action* /(1,I,N)		
(1,N,N)		*No Action* /(0,N,N)	Request Irrevoc.; Stall Own Transact. /(1,I,NI)
(1,I,N)		*No Action* /(0,I,N)	Request Irrevoc.; Stall Own Transact. /(1,I,NI)
(1,I,I)		BCast TS=N /(0,N,N)	
(1,I,NI)			

States	Network Events					
	Conflict w/ XS=N	Conflict w/ XS=I	Conflict w/ XS=NI	Irrevocability Granted	Broadcast TS=N	Broadcast TS=I
(0,N,N)					*No Action*	*No Action* /(0,I,N)
(0,I,N)					*No Action* /(0,N,N)	*No Action*
(1,N,N)	Normal Conflict Mgment.		Normal Conflict Mgment.		*No Action*	*No Action* /(1,I,N)
(1,I,N)	Normal Conflict Mgment.	Rollback Own Transact.	Normal Conflict Mgment.		*No Action* /(1,N,N)	*No Action*
(1,I,I)	Rollback Other Transact.		Rollback Other Transact.			
(1,I,NI)				BCast TS=I; Unstall Own Transact. /(1,I,I)	Request Irrevoc.	*No Action*

Request Irrevoc.: Request irrevocability for own transaction
Stall/Unstall Own Transact.: Stall/unstall transaction of the thread requesting irrevocability
Normal Conflict Mgment.: Default conflict management for non-irrevocable transactions
BCast TS=N (TS=I): Broadcast to all threads the command to change their state (TS) to N (I)
Rollback Own/Other Transact.: Rollback the own or the other transaction in the case of a conflict
†States (see Table 1): (0,TS,TX) → out of transaction, (1,TS,TX) → within transaction
‡Protocol transitions: <action>;<action>.../<final state>

the irrevocable transaction the former must abort since the irrevocable one cannot be aborted. Note that normal transactions (XS=N) running in threads with TS=I can commit in parallel with an irrevocable transaction, provided that they do not conflict and they do not go beyond the signature saturation threshold. This behaviour enhances the parallelism of transactions with unsaturated signatures, although it increases contention. Finally, non-transactional

Table 3 Protocol for irrevocability with exclusion

States	Thread/Core Events		
	Transaction Begin	Transaction Commit	Signature Saturation
(0,N,N)	*No Action* /(1,N,N)		
(0,I,N)	Stall Own Transact. /(1,I,N)		
(1,N,N)		*No Action* /(0,N,N)	Request Irrevoc.; Stall Own Transact. /(1,I,NI)
(1,I,N)			
(1,I,I)		BCast TS=N /(0,N,N)	
(1,I,NI)			

States	Network Events					
	Conflict w/ XS=N	Conflict w/ XS=I	Conflict w/ XS=NI	Irrevocability Granted	Broadcast TS=N	Broadcast TS=I
(0,N,N)					*No Action*	*No Action* /(0,I,N)
(0,I,N)					*No Action* /(0,N,N)	*No Action*
(1,N,N)	Normal Conflict Mgment.		Normal Conflict Mgment.		*No Action*	Stall Own Transact. /(1,I,N)
(1,I,N)					Unstall Own Transact. /(1,N,N)	*No Action*
(1,I,I)	Rollback Other Transact.		Rollback Other Transact.			
(1,I,NI)				BCast TS=I; Unstall Own Transact. /(1,I,I)	Request Irrevoc.	*No Action*

Request Irrevoc.: Request irrevocability for own transaction
Stall/Unstall Own Transact.: Stall/unstall transaction of the thread requesting irrevocability
Normal Conflict Mgment.: Default conflict management for non-irrevocable transactions
BCast TS=N (TS=I): Broadcast to all threads the command to change their state (TS) to N (I)
Rollback Own/Other Transact.: Rollback the own or the other transaction in the case of a conflict
†States (see Table 1): (0,TS,TX) → out of transaction, (1,TS,TX) → within transaction
‡Protocol transitions: <action>;<action>.../<final state>

code is allowed to run in parallel with irrevocable transactions although strong isolation is guaranteed.

3.2 Irrevocability with exclusion

The other irrevocability mechanism we have analyzed in this work implies exclusion of other transactions running in the system. A priori, it would seem that this approach hinders parallelism exploitation, but we will see in Section 5.4 that performance loss is negligible in most cases and other collateral benefits spring in terms of energy and network contention.

Table 3 shows the protocol state machine for irrevocability with exclusion. The scheme is similar to that of irrevocability with no exclusion but now normal transactional code is not allowed to run in parallel with the irrevocable transaction. Note that the scheme, however, allows non-transactional code execution. When a normal thread is running a normal transaction (state (1,N,N)) and receives the command to change its state to TS=I, it stalls the transaction. However, if the thread is running non-transactional code (state (0,N,N)), it changes its state to TS=I and continues execution. When a thread in state (0,I,N) attempts to begin a transaction, it is immediately stalled. On irrevocable mode the conflict manager changes so that the stalled transactions are aborted whenever the irrevocable transaction conflicts with them. When the irrevocable transaction commits, it broadcasts the command to change the state of the other threads to normal (TS=N) and they resume execution.

3.3 Example of operation

Fig. 4a shows an example of operation of the irrevocability mechanism described in Section 3.1. We have three threads, Th 0-2, running non-transactional code, OutXact. First, Th 1 begins a transaction followed by Th 2. Both happen to saturate the signature at the same time, so they move to the transient state (TS=NI) and irrevocability is granted to one of them (Th 1). Th 2, then, waits stalled until the irrevocable transaction in Th 1 commits. Th 1, on the other hand, sends the command to set the state of all threads to TS=I and resumes execution of the transaction as irrevocable (XS=I).

In the meantime, Th 0, which is in state TS=I, begins a normal transaction (state XS=N). It is allowed to continue, although a conflict with the irrevocable transaction makes it abort and restart the normal transaction. Once the transaction in Th 0 restarts, the irrevocable transaction in Th 1 commits and all threads move to normal state (a command to set TS=N is broadcast by Th 1 to all threads). Then, Th 2, whose transaction was stalled due to signature saturation, requests for irrevocability again, which is granted this time. Th 2 sets all threads to state TS=I and resumes execution of the transaction in irrevocable mode.

Fig. 4b depicts an example of the operation of the irrevocability mechanism with exclusion. Now, we have the same three threads but only the transaction in Th 1 reaches the saturation of its signature. Irrevocability is granted to such a transaction and all threads move to TS=I state. Th 0 is out of transaction so it continues the execution. However, Th 2 is running a transaction so it has to

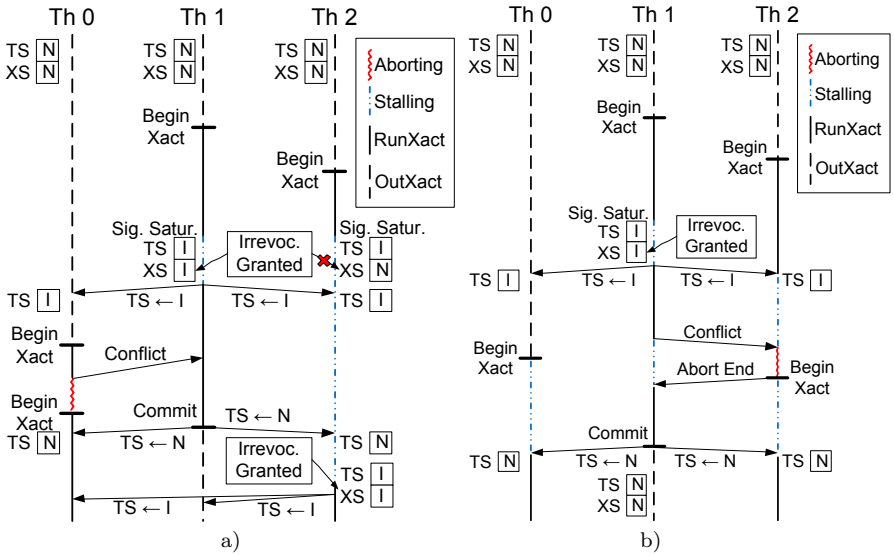

Fig. 4 Examples of irrevocability: a) With no exclusion, b) With exclusion.

stall because this irrevocability mechanism excludes all transactions running in the system except the irrevocable one.

After a while, Th 0 begins a transaction, but it stalls immediately due to exclusion (TS=I). Also, Th 1 conflicts with the stalled transaction in Th 2 so such transaction in Th 2 is aborted. In this scenario the TM system implements eager version management, therefore Th 1 stalls until the abort is done and the conflicting data released. In case of lazy version management, it is not necessary to stall the transaction. Finally, the irrevocable transaction (in Th 1) commits and the other transactions resume execution in normal mode (TS=N).

3.4 HTM implementation

The described irrevocability schemes were implemented in a HTM system simulator (see Section 5.1), where a token-based distributed mechanism was used for granting irrevocability. Irrevocability is presently included in commercial systems [37], but implemented as a runtime. In our case, the token-based mechanism used in our simulations introduces negligible overhead (see Section 5.3) and has some benefits over a software one (see Section 3.6).

The baseline architecture used in this paper is a tiled multicore processor where each tile (core) comprises a CPU, a split primary instruction and data cache, and a shared banked secondary cache (see Fig. 5). Cores are connected each other by means of a point-to-point grid interconnect which is accessed by L1 and L2 cache controllers via request and response queues [35]. In addition to the hardware required to support transactional execution (see Section 5.1),

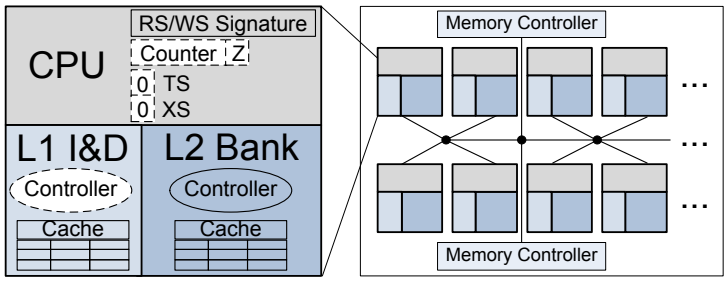

Fig. 5 Baseline architecture and hardware requirements (in dashed lines).

each core is augmented with a Counter (that includes a zero hardware signal, Z) and two 1-bit flags that hold the thread (TS) and transaction (XS) states.

Counter keeps count of the number of events that are tracked to signal the signature saturation condition (Sections 5.2 and 5.3). Initially, it is set to a *threshold* value which might be fixed or configured. Such a value could be hard-wired depending on the size of the signature and the amount of false positives allowed (a hard-wired threshold is a good solution, see Section 5.3). When an event is detected within the boundaries of a transaction the counter is decremented by 1. If the counter is 0, then the signal Z is raised, indicating that the signature is about to saturate. Then, if Z=1 and the transaction is granted irrevocability (arbitrated by the token-based mechanism), the thread sets its XS flag to I and broadcasts the command to set TS flag to I to the rest of the threads.

In order to communicate any TS changes to other threads and to implement the token-based arbitration mechanism, new message types have to be added to the *L1 Cache Controller* protocol. However, there is no need to change the cache coherence protocol, which is a critical element of a multicore processor, difficult to specify/verify [35]. These new messages are control messages and they are not related to addresses unlike coherence messages. Therefore, as such messages do not include addresses, they are short packets that comprise the message type, the source and the destination processor ID.

3.5 Qualitative Comparison

OneTM mechanisms are similar to our irrevocability proposals. However, they are devised for a different baseline TM system and address a different problem. A quantitative comparison would not be fair as OneTM is based on a bounded HTM. Table 4 summarizes the main differences between them qualitatively. OneTM bounded HTM system relies on a permissions-only cache that keeps track of coherence permissions (not data) of L1-evicted transactional cache blocks to enlarge the footprint of transactions. Our irrevocability mechanisms, though, run on top of an unbounded signature-based HTM system where transactions never overflows the hardware resources. So the event that causes irrevocability is different each other. When a transaction overflows

Table 4 Characteristics of our irrevocability proposals versus OneTM approaches.

	OneTM Serialized	**Irrevoca. Exclusion**	**OneTM Concurrent**	**Irrevoca. no Exclusion**
Baseline TM system	Bounded	Unbounded	Bounded	Unbounded
Irrevocability Event	Overflowed Transacts	Signature Saturation	Overflowed Transacts	Signature Saturation
Stalls other transactions	Yes	Yes	No	No
Stalls non-transact code	Yes	No	No	No
Hardware requirements	Per-thread Stat Words	Per-thread bits/counter	Per-block Metadata	Per-thread bits/counter
Implementation	Unspecified	Cache Controllers	Controllers	Cache Controllers
Cache coherence mods	No	No	No	No

the hardware resources in OneTM it becomes irrevocable to ensure forward progress. In our system, such an event is signature saturation, and we use irrevocability to gain performance as the execution always comes to and end.

OneTM-Serialized stalls both transactional and non-transactional code as the overflowed transaction does not keep track of overflowed data. Thus strong isolation cannot be enforced, whereas our signatures still keep the RS/WS of the irrevocable transaction and non-transactional code can persist. Concerning hardware requirements, OneTM needs a per-thread status word and a shared transaction status word. The latter encodes an overflowed bit and the ID of the overflowed transaction. Such a word is used as a lock to enter overflowed mode. However, they do not specify whether the threads access the status words by software or hardware. OneTM-Concurrent also needs per-block metadata architected in caches and memory so that transactional information of the overflowed transaction comes along with the data, with the subsequent hardware complexity. Memory controllers are modified to deal with the additional metadata. Our distributed irrevocability proposals use per-thread bits and counter as seen in Section 3.4. The protocols are implemented as state machines in the cache controllers. Neither of the proposals need modifications of the cache coherence protocol.

3.6 Software Alternatives

There are two main software alternatives to our hardware irrevocability mechanisms: a fallback path or a runtime, such as that of Blue Gene/Q. Fig. 6 shows the code for the fallback path alternative. The user is burdened with the task of writing such a fallback code, as in best-effort HTM systems [21], which complicates the TM paradigm.

When using a fallback path, the hardware should detect the saturation scenario, abort the saturated transaction and set a special register (line 1) with the cause of abort. The register is read by the fallback code to determine whether to serialize or not (line 11). Line 3 begins the transaction and, in case of abort, returns the transactional register into variable `treg`. A common way

```
 1  tregisterType treg;
 2  while (true) { // Infinite loop
 3    if (treg = tbegin()) { // Transactional code
 4      // Successful transaction begin
 5      if(globalLock) tabort(); // Lock subscription
 6      // Transaction body
 7      tcommit();
 8      break; // Transaction committed. Break while loop
 9    } else { // Non-transactional code
10      // Transaction abort
11      if (treg == SIGNATURE_SATURATION) { // Abort due to signature saturation
12        acquireLock(globalLock);
13        // Transaction fallback
14        releaseLock(globalLock);
15        break;
16      } // Transaction fallback finished. Break while loop
17      // Retry transaction
18    }
19  }
```

Fig. 6 Software alternative code to our irrevocability proposals.

to code the fallback path comprises a global lock to execute the transaction as a non-transactional critical section (lines 9 to 18). In addition, when a transaction begins, the fallback global lock is checked (lock subscription, line 5). If the lock is acquired, the transaction aborts. If not, the transaction goes ahead with the lock in its read set so that another transaction acquiring the lock can abort it. Interleaving transactions and fallback path sections is thereby avoided to maintain consistency between transactional and non-transactional accesses (no exclusion is not possible). This fallback behaviour can be detrimental to performance as we show below.

A runtime, though, implements the irrevocability alternative without user intervention (the code in Fig. 6 would be generated by the compiler). However, it is implemented with a lock like that of the fallback path, entailing the same performance pitfalls. Fig. 7 shows an execution scenario comparing our irrevocability with no exclusion with a software lock-based alternative (fallback or runtime). Our mechanism allows parallelism in the absence of conflicts. Th1's signature saturates and requests irrevocability. Irrevocability is granted and the transaction continues irrevocably. Th2's transaction commits before Th1 becomes irrevocable, and Th0's transaction is allowed to execute in parallel. With the software lock-based alternative (fallback or runtime) the transaction in Th1 aborts when saturation is detected. Previously, every transaction has subscribed to the lock in order to be aborted whenever a transaction acquires the lock, so that transactional and non-transactional lock-protected code (Run-Locked) do not coexist. Thus, lock acquisition in Th1 causes Th0's transaction abort. Th2's transaction commits before lock acquisition, so it is not aborted. Th0 waits for the lock to be released and then it begins the transaction again. The performance is therefore hindered.

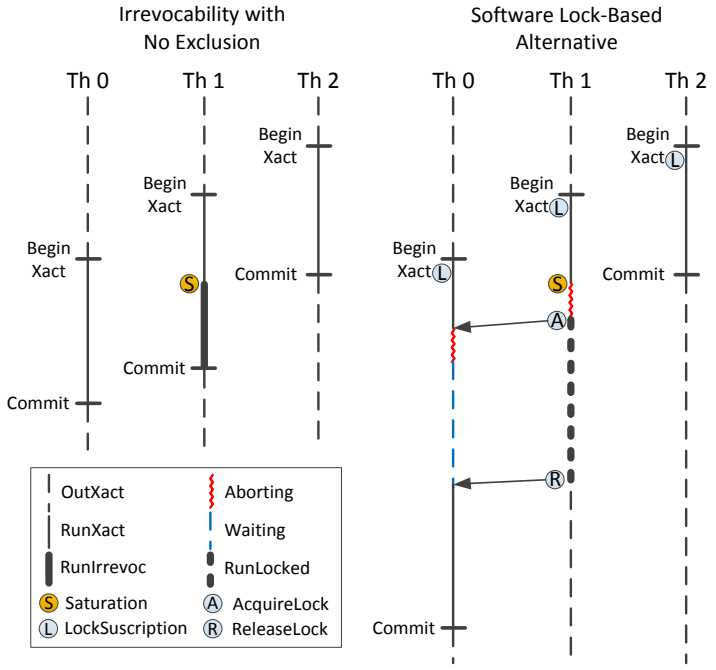

Fig. 7 Execution scenario comparing our irrevocability with no exclusion mechanism with a software lock-based alternative implementation.

4 Analytical Model

In this section, we propose an analytical model approximation for a TM system with irrevocability considering the following premises: (i) the normal irrevocability mechanism of Section 3.1 is analyzed; (ii) a lazy conflict detection mechanism is assumed, which implies that aborts can arise only after the entire execution of transactions; (iii) neither commits nor aborts are modeled, so the increase of transactional execution time comes from the repetition of transactions; (iv) a transaction is aborted on conflict (stall is not modeled). These constraints simplifies the model.

Fig. 8 shows how we have approached the analysis. The execution time of the system with irrevocability, T, can be worked out as the sum of the parallel time of all transactions, T_p, included those that are greater than the signature saturation threshold (θ) until they reach θ, and the serial time of the chunks of transactions that exceeds θ, T_i. So, let M be the number of transactions to be executed by N threads, with $M \geq N$; let $p_i = \sum_{sz \geq \theta} Pr(sz)$ be the probability of a transaction becoming irrevocable, given a probability distribution function for the size of transactions, $Pr(X)$, defining the size as the cardinality of the data set (RS∪WS); let $\bar{t}_i$, $\bar{t}_c$ and $\bar{t}_a$ be the expected execution time of the chunks of transactions that exceed θ, the transactions that successfully commit in parallel, and the transactions that abort, respectively; and let N_r be the

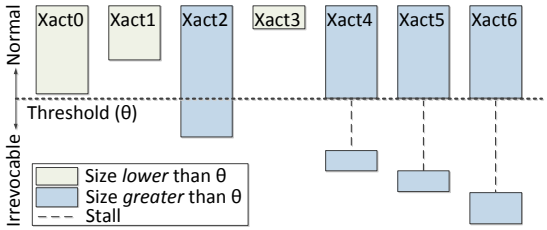

Fig. 8 Irrevocable and normal mode differentiation for the analytical model.

number of transaction retries. Then, T can be expressed as:

$$T = T_i + T_p = p_i M \bar{t}_i + \frac{M}{N}(\bar{t}_c + \bar{t}_a(N_r - 1)). \tag{1}$$

We have considered that the execution time of a transaction is dependent on its size, $t_{\text{cpu}}(sz)$. Then, the expected execution time of the transactions' chunks that exceed the threshold θ, given the probability distribution function for the size of transactions $Pr(X)$, is given by the following equation:

$$\bar{t}_i = E[t_i] = \frac{\sum_{sz \geq \theta} Pr(sz)(t_{\text{cpu}}(sz) - t_{\text{cpu}}(\theta))}{\sum_{sz \geq \theta} Pr(sz)} \tag{2}$$

The expected time of the transactions that successfully commit in parallel can be determined by:

$$\bar{t}_c = E[t_c] = \sum_{sz} Pr(sz) t_{\text{cpu}}(min(sz, \theta)). \tag{3}$$

Note that transactions larger than the threshold are split in two parts, as Fig. 8 shows. The part of them that lies beyond the threshold is modeled with t_i in Eq. 2. The part that lies before θ, along with those transactions with size lower than θ, are included in Eq. 3. Therefore we use the min function.

For the expected time of aborting transactions, $\bar{t}_a$, we calculate the average size of the transactions that eventually abort and we get the execution time:

$$\begin{aligned} \bar{t}_a &= E[t_a] = t_{\text{cpu}}(\overline{sz}_a) \\ \overline{sz}_a &= \frac{\sum_{sz} Pr(sz) min(sz, \theta) \overline{p}_a(sz)}{\sum_{sz} Pr(sz) \overline{p}_a(sz)}, \end{aligned} \tag{4}$$

where $\overline{p}_a(sz)$ is the probability that a transaction of size sz finds a conflict and aborts consequently. The conflict can occur with any of the $N-1$ transactions running in the system, so $\overline{p}_a(sz)$ is defined in this way:

$$\overline{p}_a(sz) = E[p_a(sz)] = 1 - (1 - \overline{p}_c(sz))^{N-1}, \tag{5}$$

with $\overline{p}_c(sz)$ being the average conflict probability of a transaction of size sz:

$$\overline{p}_c(sz) = \sum_{sz'} Pr(sz') p_c(min(sz, \theta), min(sz', \theta)). \tag{6}$$

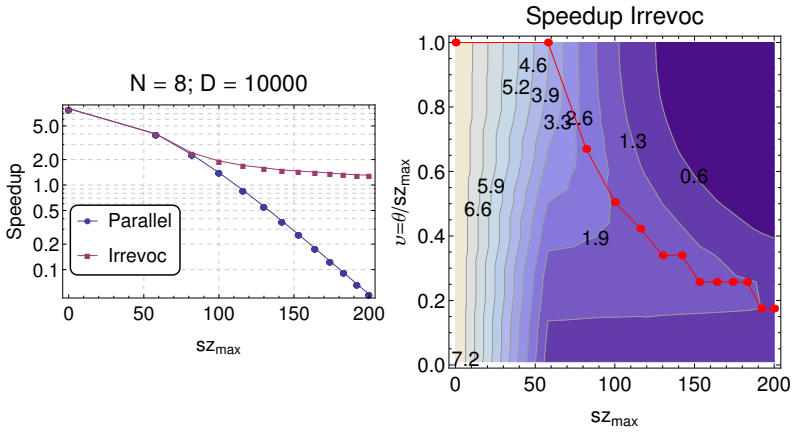

Fig. 9 Left: speedup of parallel and irrevocable systems with 8 threads and 10K shared locations. Right: iso-speedup curves of the irrevocable system varying sz_{max} and θ. The red line shows the best θ.

The probability of conflict between two transactions of sizes sz and sz', $p_c(sz, sz')$, according to [44] is:

$$p_c(sz, sz') = 1 - \frac{\binom{D-sz}{sz'}}{\binom{D}{sz'}}, \tag{7}$$

where D is the cardinality of the set of locations accessed by all transactions in the system, the shared locations.

Finally, transactions may abort several times before committing. The average number of transaction retries, Nr, is given by:

$$\begin{aligned} N_r &= \sum_{i=1}^{\infty} i\ \overline{p}_a{}^{i-1}(1-\overline{p}_a) = \frac{1}{1-\overline{p}_a} \\ \overline{p}_a &= 1-(1-\overline{p}_c)^{N-1} \\ \overline{p}_c &= \sum_{sz} Pr(sz)\overline{p}_c(sz). \end{aligned} \tag{8}$$

Note that $\overline{p}_a$ and $\overline{p}_c$ are the generic counterparts of Eqs. 5 and 6. $\overline{p}_c$ uses $\overline{p}_c(sz)$ of Eq. 6 as well.

Fig. 9 shows the speedup of the parallel and the irrevocable systems with 8 threads and 10K shared locations (left graph). We deem the size of transactions uniformly distributed from 0 to sz_{max}, and the transaction execution time as a linear function of the size, $t_{\text{cpu}}(sz) = \alpha sz$. The Irrevoc line represents the speedup of the system with the best choice for the irrevocability threshold θ. We can see that the parallel system is worse than serial one from $sz_{max} = 110$ onwards, because of the number of transaction retries. However, the system with irrevocability tends to settle close to sequential without performing worse than it. The graph on the right of Fig. 9 shows the iso-speedup

Table 5 Workloads: Input parameters and TM characteristics

Bench	**Input**	**# xact**	**% time in xact**
Bayes	-v32 -r1024 -n2 -p20 -i2 -e2 -s1	624	98%
Genome	-g512 -s64 -n16384	30304	93%
Intruder	-a10 -l64 -n2048 -s1	91374	94%
Kmeans	-m15 -n15 -t0.05 -i random-n1024-d1024-c16	2760	15%
Labyrinth	-i random-x32-y32-z3-n48	126	100%
Vacation	-n16 -q60 -u90 -r16384 -t4096	4095	97%
Yada	-a20 -i633.2	5286	100%

Bench	**XactID(avg|**RS**|/avg|**WS**|)**
Bayes	0(2/1) 1(22/3) 2(6/3) 3(22/11) 4(53/20) 5(2/1) 6(74/33) 7(34/12) 8(4/1) 9(484/274) 10(55/34) 11(75/44) 12(21/3)
Genome	0(507/9) 1(4/1) 2(8/4) 3(6/4) 4(6/3)
Intruder	0(4/1) 1(36/6) 2(2/1)
Kmeans	0(134/65) 1(1/1) 2(2/1)
Labyrinth	0(3/1) 1(160/223) 2(7/3)
Vacation	0(149/14) 1(33/6) 2(132/14)
Yada	0(8/2) 1(1/0) 2(351/225) 3(1/1) 4(9/2) 5(2/2)

curves of Irrevoc varying the maximum limit, sz_{max}, of the transactions' discrete uniform distribution, and the ratio θ to sz_{max}, υ, that tells where the irrevocability threshold is placed. The line shows the choice of θ that yields the best system performance. When transactions are small the conflict rate is low and irrevocability is not needed, $\upsilon = 1$. However, a low value of υ is needed when transaction size increases to avoid multiple transaction retries.

5 Experimental Evaluation

In this section we discuss the experimental results obtained from evaluating the irrevocability mechanisms described in Section 3. We explore different saturation metrics to trigger irrevocability (Sections 5.2 and 5.3) and different signature designs (Section 5.5). We also discuss the impact of the irrevocability mechanisms in the network interconnect power and traffic (Section 5.4).

5.1 Methodology

We have used the Simics [22] full system simulator, together with the Wisconsin GEMS [23] module for Simics. GEMS includes an implementation of the LogTM-SE HTM system [41] in its Ruby module that we have modified to include the irrevocability schemes.

The target system is organized as shown in Fig. 5. It comprises 16 in-order single-issue cores, with a private 32KB split L1 cache. L2 cache is unified, shared and divided into 16 banks of 512KB each. L2 is 8-way associative with 64B blocks. The directory keeps a full bit-vector of sharers. For the network

time and power modeling we have used Garnet [1], a detailed interconnection model that integrates Orion [19], a network power model. The interconnect has 128bit flits, 4 virtual channels per virtual network, and 4 flits per buffer. The number of ports per router is variable depending on the position of the router in the network. We configured Orion with a 32nm process, NVT transistors and 1.0 Vdd.

We have used parallel Bloom filters with four $H3$ hash functions to implement signatures. Enhanced signature schemes like Unified [11] or Multiset Locality-Sensitive [27,28] are also evaluated. Filter size ranges from 64 bits, which matches the architecture word length, to 8K bits length, which matches the performance of perfect signatures (hardware unimplementable structures that do not yield false positives) for the evaluated benchmarks. The number of threads is 15 as one core is left to the operating system.

Lastly, Ruby adds pseudo-random delays to memory accesses to deal with variability in simulation experiments. Then, multiple runs of each experiment were carried out to obtain confident error bars [2].

Benchmarks

Regarding the benchmarks, all the codes of the Stanford's STAMP suite [24] were used for experimentation, except SSCA2, which has been proved to be signature-insensitive [28] because of its small (about 3 blocks on average) and short time transactions. The other benchmarks in the suite exhibit long-running transactions and relatively large transactions as shown in Table 5. Some of the benchmark input parameters were set to enlarge transaction size without increasing noticeably the duration of the simulation (e.g. CHUNK_STEP1=64 for Genome), since STAMP recommended large inputs are very demanding for the simulation. The table shows the number of transactions that successfully commits "# xact", the percentage of time running transactions "% time in xact", and the average RS/WS cardinality for each transaction, in cache blocks. Each transaction is identified by its ID, starting from 0 and numbered in order of appearance in the code.

With respect to Labyrinth, the optimized version of the benchmark [39] requires the use of *early release* [15], a feature that allows the user to remove random read locations from the signature. However, removing locations from a Bloom filter is not allowed, so we implemented the early release feature by resetting the entire read filter. In the case of Labyrinth, it does not affect correctness since the early release is performed at the beginning of the transaction, so a true early release would produce the same result. However, such an early release implementation is not valid for selective read location deletion.

Irrevocability threshold metrics

In the following sections, we show the results obtained for two different metrics of the size of a transaction. We have tracked events that measure the size of the transaction to trigger irrevocability, as the larger the transaction the

Table 6 Number of transactions that overflow the L1D cache.

Benchmark	Overflowing Transactions	Avg # of xact blocks replaced (Only Read/Written)
Bayes	102	68.2/100.8
Genome	447	78.7/1.8
Intruder	4511	2.1/0.1
Kmeans	387	1.0/0.0
Labyrinth	48	62.9/76.8
Vacation	2710	7.0/0.1
Yada	816	117.2/73.2

more saturated the signature. Specifically, we evaluate both the cache overflow count, which stands for the number of transactional block replacements during a transaction, and the transactional load and store count.

5.2 Cache overflow as irrevocability threshold

The number of transactional blocks replaced from the L1D cache can be considered as a measure of the amount of data accessed by a transaction. Actually, best-effort HTM systems, such as Intel Haswell, abort whenever there is a transactional replacement, called a data capacity limitation. Thus, these systems only manage small transactions entirely in hardware. Haswell provides performance counters (TX_MEM and TX_EXEC [30]) to track these events.

In this case, we consider a counter of L1D cache replacements of transactional blocks as irrevocability threshold. Table 6 shows the number of transactions that overflow the L1D cache (i.e. replace a transactional block) and the average number of transactional block replacements. We can see that all the STAMP benchmarks tested exhibit overflowing transactions and all of them replace one or more transactional cache blocks during the execution of a transaction. Therefore, the benchmarks could not be successfully executed in a best-effort HTM system without programming a fall-back code to deal with data capacity limitations. Note that overflowing transactions and the average number of transactional blocks replaced by them have not to be necessarily correlated with each other. Bayes has a few large transactions, whereas Intruder exhibits many transactions that overflow the cache by replacing just two or less transactional blocks.

As regards the threshold, we can choose to switch to irrevocable mode when the first replacement takes place, regardless of the signature size. However, we found that making a transaction irrevocable too early can limit useful parallelism. Then, we evaluate a policy that comprises a threshold fraction of the signature size. Fig. 10 shows the execution time normalized to perfect signatures (lower is better) for the sequential version of the benchmark and the base TM system running the parallel application. It also shows the results of the irrevocability mechanism (with no exclusion) using cache overflow as threshold (OvFlow). We vary the threshold from 1/128th up to 1/32nd the signature size, so that transactions get irrevocable upon the first replacement

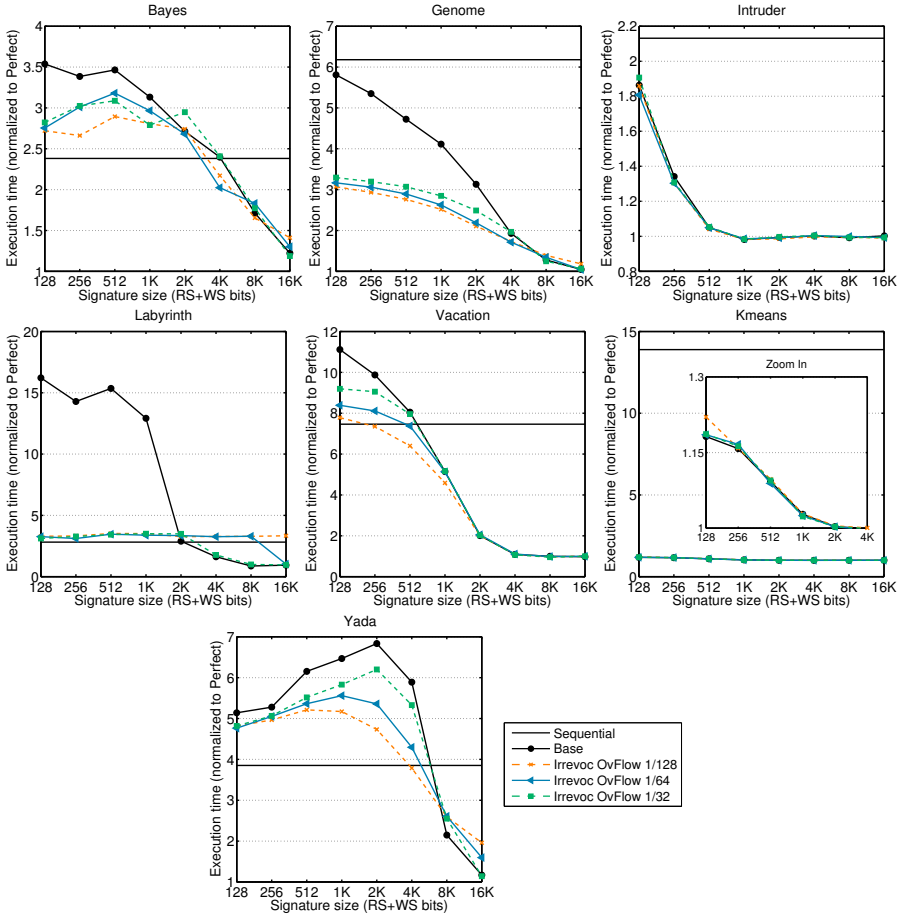

Fig. 10 Results of irrevocability using cache overflow as threshold (OvFlow). Threshold varies from 1/32nd up to 1/128th the signature size. Sequential and Base TM system results are shown for comparison.

when signature is 128bit and irrevocability is delayed until 512 replacements when signature is 16Kbit.

We can see that this method of measuring transaction size is useful for certain benchmarks like Genome, Labyrinth and Vacation. However, other benchmarks are less sensitive to this policy, like Bayes or Yada. Intruder and Kmeans perform the same as the base system.

Bayes and Yada, although they perform better than the base TM system in most cases, they seem to need an earlier irrevocability to get nearer to sequential when signatures are small. Note that, even with the minimum threshold that corresponds to OvFlow 1/128 and 128bit signatures, Bayes and Yada are closer to the base system than to the serial execution. This is a result of the late appearance of the first replacement that postpone irrevocability.

On the other hand, Genome, Labyrinth and Vacation exhibit very good results depending on the threshold. OvFlow 1/32 gets the best results for Labyrinth, whereas OvFlow 1/128 is the best configuration for Vacation and Genome, since it yields the same as or much better performance (see Genome performance improvement discussion in next section) than sequential while Base performs worse. Also, execution is not harmed when signatures are large enough to allow a profitable parallel execution.

As aforementioned, signatures can lead the parallel version of an application to perform worse than the serial one. This can be seen in Fig. 10 as the signature size decreases. Base performance falls down beyond that of Sequential for almost every benchmark except for Kmeans and Intruder. Although Kmeans exhibits large transactions on average, it spends short time in transactions (see Table 5) and gets a high speedup difficult to degrade to such an extent. Intruder spends most of the time in transactions but these are relatively short both on average and maximum values.

The advantages and disadvantages of using cache overflow as irrevocability threshold are the following:

- *Pros*: For some benchmarks, the overflow policy do not harm the performance for large signatures, whereas it reaches the goal of performing like the sequential version when signatures are short. The overflow policy can even outperform the sequential version in certain circumstances.
- *Cons*: The lowest threshold possible is one replacement, which can be insufficient for benchmarks with a late replacement appearance or that need an earlier irrevocability. Also, there is not a single threshold that performs equally well for all the benchmarks tested, and this makes it difficult to implement the scheme in a general purpose system without adding a mechanism to determine the proper threshold for an application.

5.3 Using load and store count as irrevocability threshold

A more direct way to measure the size of a transaction is to count its loads and stores. However, a problem arise when it comes to counting insertions into the signature. One method would decrement the counter on each insertion. But, load and store repetitions due to temporal locality could decrement it too fast and cause an early irrevocability. We propose a load/store count policy that comprises an insertion operation that checks whether the address was previously inserted or not. If not, the counter is decremented. Conventionally, insertion operations always insert the address into the signature, regardless of whether the address is already inserted or not. Since signatures are implemented as SRAMs, our proposal could use a read/modify/write SRAM cycle (such as that used in caches to set the LRU bits) for the insertion operation without compromising performance significantly, where the address is read early in the cycle and written late in the cycle if applies.

Counting insertions trusting the signature can lead to miscounting due to false positives. If we inserted an address X that is alias of an address Y and

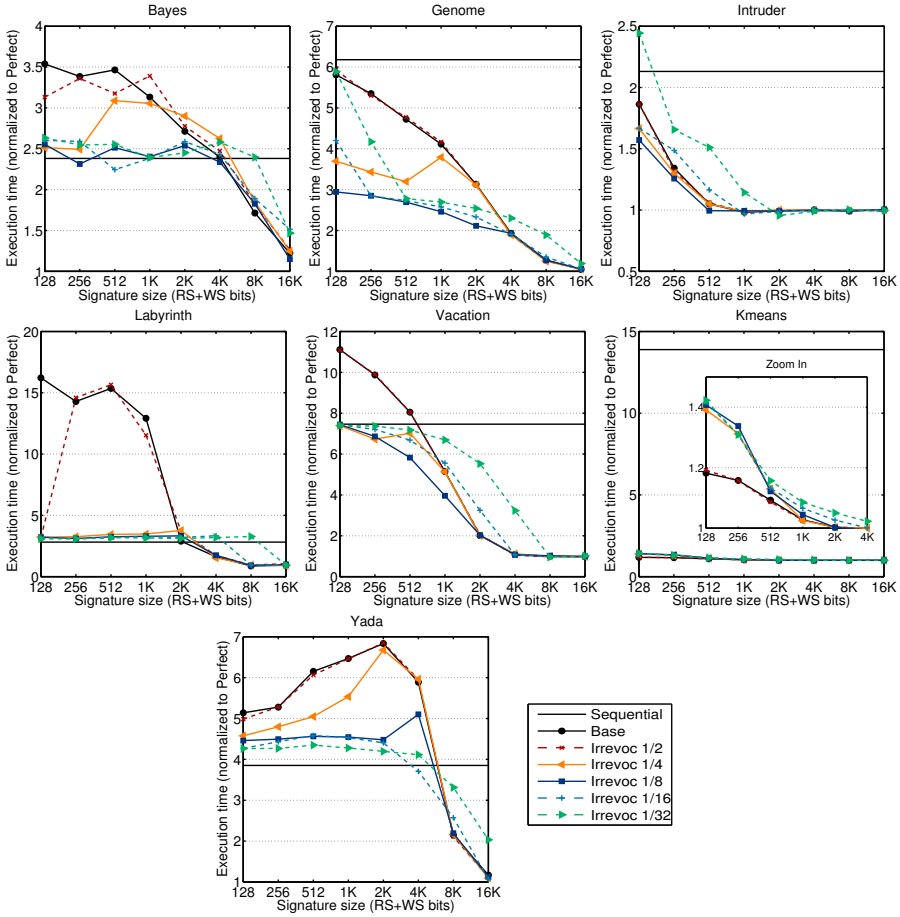

Fig. 11 Results of irrevocability using transactional load and store count as threshold (Irrevoc Count). Threshold varies from 1/2nd up to 1/32nd the signature size.

then we insert Y into the filter, Y will hit the signature and the counter will not be decremented. Then, irrevocability mode might not be triggered when the false positive rate is too high. Note that false positives and irrevocability threshold are dependent each other. The higher the threshold the more false positives. However, we will see here that a low threshold is a good choice, so the miscounting problem is negligible.

Also, we have considered only one counter for both RS and WS signatures, as [11,27] suggest that unifying in terms of signatures is a good choice to deal with asymmetry in data sets.

Fig. 11 shows the execution time normalized to perfect signatures (lower is better) of the serial application, the base TM system (Base) and the system with the irrevocability scheme (with no exclusion) and the count threshold (Irrevoc Count). The threshold from which irrevocability begins is shown as

a fraction of the size of the signature (RS+WS), and it varies from a half to 1/32nd. Since the signature size ranges from 128 to 16K (abscissa axis), the minimum threshold is 4 insertions and the maximum 8K. This policy allows a finer control over the threshold than the cache overflow policy described in Section 5.2.

As far as irrevocability is concerned, we can see a trend towards Base as threshold increases. However, Irrevoc Count tends to Sequential as threshold decreases, which was expected. Presumably, a one insertion threshold ($Counter = 1$) would yield the performance of the serial application plus a slight overhead due to irrevocability control messages. This trend can be perfectly noted in Labyrinth where Irrevoc Count 1/2 is quite similar to Base and Irrevoc Count 1/32 almost matches Sequential.

When it comes to choosing the best threshold for our application, this choice can be determined in different ways (as discussed in Section 3.4): on a per-application basis, on a per-transaction basis or with a hard-wired threshold. In this case, we tested several thresholds on a per-application basis and we found that a threshold of 1/8th the signature size yields the best results for all benchmarks except for Kmeans, which get the best performance with 1/2nd (Irrevoc Count 1/2 gets the same results than Base). However, we would get a negligible loss of performance by choosing Irrevoc Count 1/8 for this benchmark, and it does not get worse results than its sequential version in any case. So, a hard-wired threshold can be a good option, since it would not be necessary to augment the ISA with an instruction for setting the threshold, thus hiding the difficulties for a programmer (or compiler) to deal with a new instruction and enhancing the portability of transactional code.

The theoretical maximum percentage of false positives that we can obtain with a threshold of 1/8th the signature size is given by the equation [26]:

$$p_{\text{FP}}(M, n, k) = \left(1 - \left(1 - \frac{1}{M}\right)^{nk}\right)^{k} \approx \left(1 - e^{-\frac{nk}{M}}\right)^{k} \tag{9}$$

where M is the signature size, n the number of insertions and k the number of hash functions. p_{FP} can be simplified by using the Taylor series expansion of the exponential function, $e^x = \sum_{n=0}^{\infty} \frac{1}{n!} x^n$ [32]. Replacing $n = M/8$ and $k = 4$ in equation (9) we obtain $max(p_{\text{FP}}) \simeq 0.023969$. Then, it is better to trigger irrevocability when signatures are yielding more than 2.4% of false positives than allowing parallel progress with increased contention due to false conflicts.

Irrevocability Mechanism Overhead

Fig. 12 shows a cycle breakdown for the benchmarks using Irrevoc Count 1/8 and 128bit signatures. To measure the overhead of our implementation of irrevocability we broke down the cycles into overhead cycles (Irrevoc OH), comprising the cycles spent in requesting and passing the token, the cycles to pass the irrevocability messages, and the cycles that a transaction spends

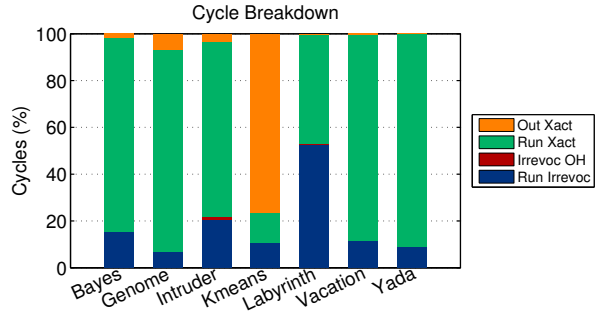

Fig. 12 Cycle breakdown for Irrevoc Count 1/8 and 128bit signatures.

stalling when it is running alone in irrevocable mode. The cycles of the transactions running in irrevocable mode is shown as Run Irrevoc. In addition, non-transactional (Out Xact) and transactional cycles (Run Xact) are shown. Note that our token-based implementation of the irrevocability mechanisms does not incur significant overhead as Irrevoc OH bars can barely be seen in the figure. Practically, the whole execution lies in the Out Xact, Run Xact and Run Irrevoc phases of the application.

So, as the overhead of the irrevocability scheme is negligible we can conclude that the reason why performance is worse or better than sequential is because of the Run Xact parallel phase. Depending on the benchmark, the transactional parallel phase may exhibit more or less contention. In the case of Labyrinth, for example, we obtain a slight performance decrease for Irrevoc Count 1/8 over Sequential, and the signature saturation threshold for signature size 128 is set to 16 insertions, so ID-1 transactions are all irrevocable (see Table 5) while XactID-0 and XactID-2 are not. XactID-0 and XactID-2 are in charge of popping a pair of coordinates from a queue and inserting a path to a global list respectively, which are conflict-prone procedures, so the phase which is not irrevocable is not helping to enhance performance.

On the other hand, Genome shows very good results when using Irrevoc Count 1/8. Genome comprises 5 transactions (see Table 5) where only the first one gets irrevocable as it goes beyond the 16-insertion threshold for 128bit signatures. The rest are small transactions whose retry count is close to zero, so the transactional phase is quite uncontended thus enhancing overall performance. Also, when comparing the irrevocability scheme to the base system, the retry count of XactID-0 reduces drastically from 4 to 1. Therefore, irrevocability maintains the benefit of small uncontended parallel transactions and limits the harm of large signature-saturating transactions in this case.

To sum up, we get a maximum of 1.3× slowdown with respect to Sequential for Yada and 4K signatures by using irrevocability with a signature threshold of 1/8th the signature size. However, we obtain up to 2× speedup over Base for Genome and 128bit signatures. Base yields a maximum of 6× slowdown over Sequential for Labyrinth and 512bit signatures, and worsens the performance beyond Sequential for most benchmarks when signatures are small.

The advantages and disadvantages of using load and store count as threshold are the following:

- *Pros*: More flexibility and determinism to configure when the irrevocability begins: the minimum threshold is not constrained (with the cache overflow policy it depends on the cache structure and the application). Also, we find that one fixed threshold of 1/8th the signature size performs evenly for the benchmarks tested.
- *Cons*: The proposed hard-wired 1/8th threshold might be considered too low as it implies a considerable amount of signature underutilization.

Implication: Load and store count is a better metric than cache overflow to estimate the size of transactions and it should be used as the event for triggering irrevocability. A fixed threshold of 1/8th the signature size can be considered as a general solution.

5.4 Irrevocability with exclusion: Reducing interconnection network power and traffic

Fig. 13 shows the results obtained for both irrevocability mechanisms, with no exclusion (Irrevoc) and with exclusion (IrrevocEx), using a signature saturation threshold of 1/8th the signature size. Base and Sequential are depicted as well. We can see that irrevocability with exclusion, IrrevocEx, settles very close to irrevocability without exclusion, with two exceptions, Bayes and Intruder.

Bayes is a benchmark that shows high variability because its performance depends on the order in which dependencies are learnt [24]. However, we can note a tendency for IrrevocEx to outperform Irrevoc when using intermediate signature sizes. The number of aborts that IrrevocEx yields is lower than that of Irrevoc for Bayes. This fact points out that Bayes shows high contention, increased by false contention introduced by the signature. Therefore, allowing transactional execution in parallel with irrevocable transactions is not always profitable.

Conversely, Intruder seems to benefit from the increased parallelism offered by Irrevoc. Intruder shows high contention in its small XactID-0 and medium contention in XactID-1, which is the coarse transaction that reaches the saturation threshold. XactID-0 and XactID-1 access disjoint data and they can be executed in parallel by different threads and successfully committed.

The rest of the benchmarks do not show much difference on using either irrevocability mechanisms. Note that if two transactions that execute the same code in parallel begin at the same time and do not conflict with each other, both would reach the signature threshold almost at the same time. Then, both mechanisms would behave similarly. One scenario that benefits Irrevoc is when there is an irrevocable transaction running in the system and the other threads are running different transactions whose data set size is lower than the signature threshold and do not conflict. Also, if the other transactions go beyond the threshold and they do not conflict with the irrevocable transaction, the transactional work done so far would be more than that with the

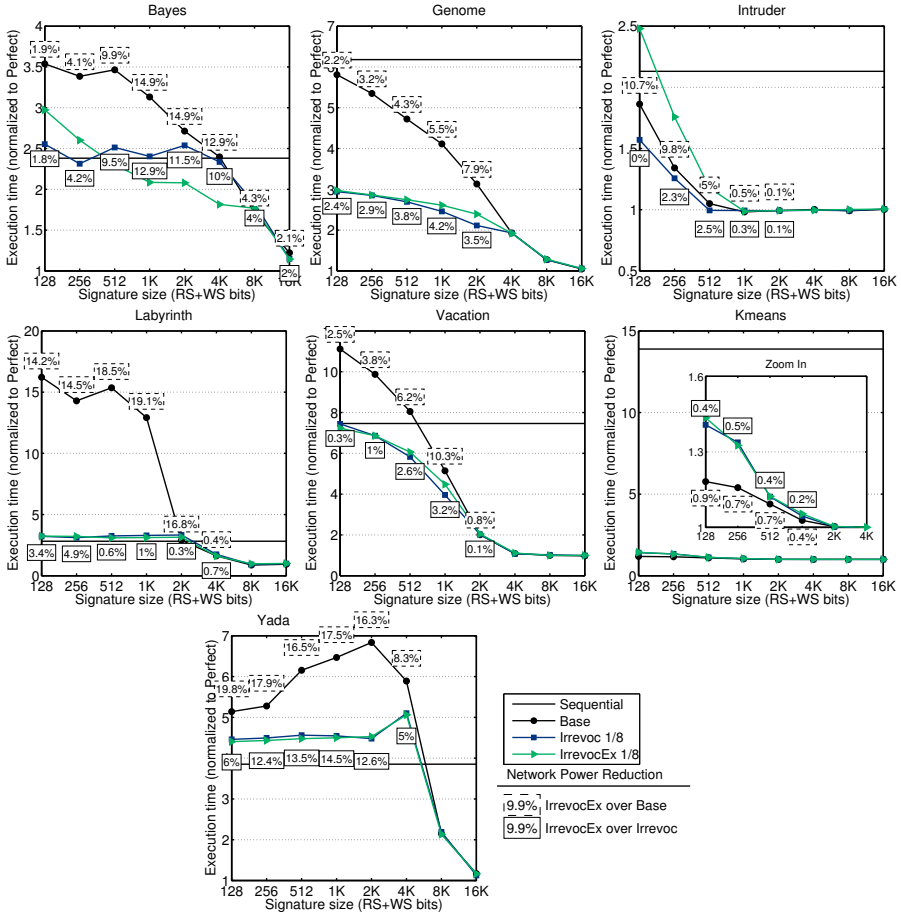

Fig. 13 Network power reduction of the irrevocability mechanisms (Count 1/8).

IrrevocEx mechanism. However, those scenarios do not show up frequently for the concerned benchmarks.

Regarding power, Fig. 13 shows the network dynamic power reduction, both for links and routers, of IrrevocEx with respect to Base and Irrevoc. Overall, both irrevocability mechanisms reduce the power requirements of the interconnect when compared with the base system. However, the reduction for IrrevocEx is significant, getting up to nearly 20% in Bayes, Labyrinth and Yada. The exclusion of all transactions running in the system, but the irrevocable one, prevent the system from consuming energy on computations that are eventually aborted because of conflicts. When compared with Irrevoc, the power reduction gain is less obvious. Around 3% for Genome, Intruder, Labyrinth and Vacation, but up to 10-15% for Bayes and Yada, which do not get benefit from the increased concurrency of Irrevoc.

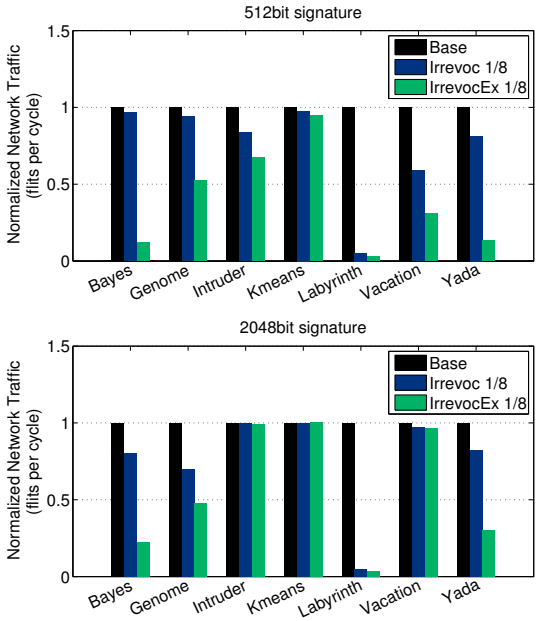

Fig. 14 Interconnect network traffic in average number of flits per cycle per link of the network. 512bit signatures at the top. 2Kbit signatures at the bottom. Traffic normalized to that of Base system.

Another benefit of the irrevocability mechanisms is their impact in network traffic. When using Irrevoc, the transactions that reach the signature threshold stall until one gets the token. On the other hand, the case of IrrevocEx is more noticeable since all transactions stall except the irrevocable one. That is, whereas in the Base TM system, stalled transactions are continuously probing the network to see if the memory location they are waiting for is available (like a spinlock), in IrrevocEx, the transactions stalled due to irrevocability are sleeping until the irrevocable transaction commits, thus not feeding the network with new messages.

Fig. 14 shows the average number of flits per cycle per link of the network for Base, Irrevoc and IrrevoEx, and two signature sizes, 512bit and 2Kbit. If the signature is large enough to deal with the transactions in the system, the irrevocability mechanism is seldom triggered and the traffic of the network is barely affected. This can be noted in Intruder, Kmeans and Vacation for 2Kbit signatures. However, the traffic reduction is substantial when the signature is 512bit. It can be seen how the IrrevocEx mechanism drastically cut the network traffic for all the benchmarks except Kmeans, which does not spend too much time in transactions (see Table 5).

Implication: The reduction in the number of messages running through the network and the subsequent energy consumption savings are important features that make the irrevocability with exclusion a mechanism worthy of

consideration. Even more when we have seen that performance is scarcely affected.

5.5 The benefit of multiset locality-sensitive signatures

Multiset locality-sensitive signatures [27] try to improve the performance of conventional signatures in two different ways. First, nearby memory locations are mapped into non-disjoint bits of the filter to decrease the occupation and, therefore, to reduce the false positive rate. They use locality-sensitive hashing [28] to take advantage of the spatial locality feature that most codes exhibit to some extent. And second, most transactional codes show asymmetric read and write set cardinalities where the read set is usually larger than the write set. To deal with asymmetry in data sets, multiset or unified signatures deploy only one filter to store both the read set and the write set, while conventional signatures have separate same-sized filters.

Fig. 15 shows the execution time normalized to perfect signatures (lower is better) of the serial application, the base TM system with multiset locality-sensitive signatures (Base MSLS) and the system with the irrevocability scheme, the load/store count policy and MSLS signatures (the rest). The threshold again is shown as a fraction of the size of the signature, and it varies from 1/2nd to 1/32nd. The specific MSLS scheme used is MS $s = 3$ L2 as described in [27].

We can see that all curves have shifted to the left, including Base MSLS, because of the multiset locality-sensitive improvement. And now, Irrevoc Count 1/8 MSLS is not the best configuration. Irrevoc Count 1/8 MSLS performs worse than Base MSLS for most benchmarks when signature size is large. This is because of early irrevocability that is triggered when the signature is still not saturated. Then, potentially useful parallel work is harmed by irrevocability instead. However, a threshold of 1/4th the signature size seems to be the best option now. Irrevoc Count 1/4 MSLS performs similar to the Base MSLS system for large signatures and outperforms it for small ones. In most cases, the execution time of Irrevoc Count 1/4 MSLS is similar to or better than the sequential whereas the Base MSLS execution time is worse.

Fig. 16 shows the results obtained for both Irrevoc and IrrevocEx, with a signature saturation threshold of 1/4th the signature size, and the use of multiset locality-sensitive signatures. Again, as discussed in Section 5.4, IrrevocEx is a great option since it yields similar performance to Irrevoc, except for a slight slowdown in Intruder and Labyrinth, and certain speedup in Bayes. However, we still get the energy improvement with up to 12% network dynamic energy savings for Bayes and Yada, and around 3% for the rest of benchmarks.

Implication: Multiset locality-sensitive signatures are a good choice to be used with the irrevocability mechanisms since the threshold can be uplifted to 1/4th the signature size, thus allowing more parallelism and better utilization of the signature's filters.

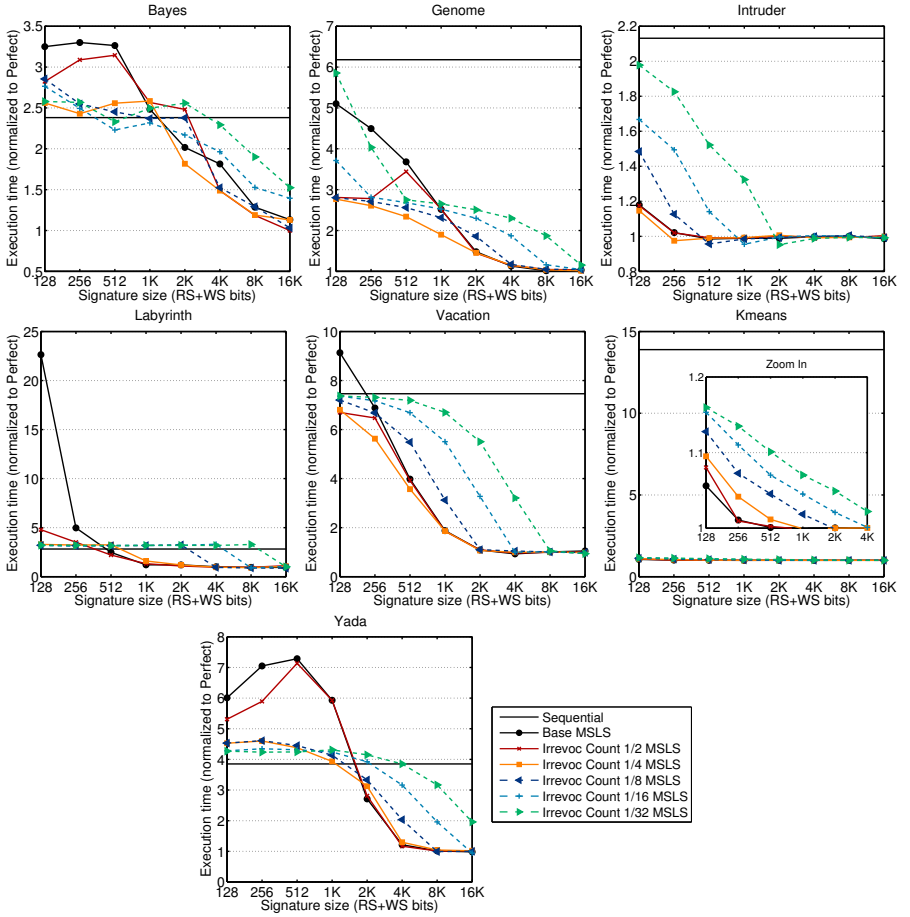

Fig. 15 Results of irrevocability using transactional load and store count as threshold and multiset locality-sensitive signatures (Count MSLS). Threshold varies from 1/2nd up to 1/32nd the signature size.

6 Conclusions

Signatures for TM have been proved to be an effective mechanism for conflict detection and TM virtualization. However, when transactions are large or signatures are small, signature false positives can exacerbate TM system pathologies to such an extent that a parallel application performs worse than its serial version.

This work presents irrevocability mechanisms to deal with such a signature saturation scenario. A transaction moves to an irrevocable execution mode when the number of insertions into the signature reaches a given threshold. Then, only such a transaction is allowed to continue while the others are stalled or left to run until they reach the threshold. The irrevocable transaction

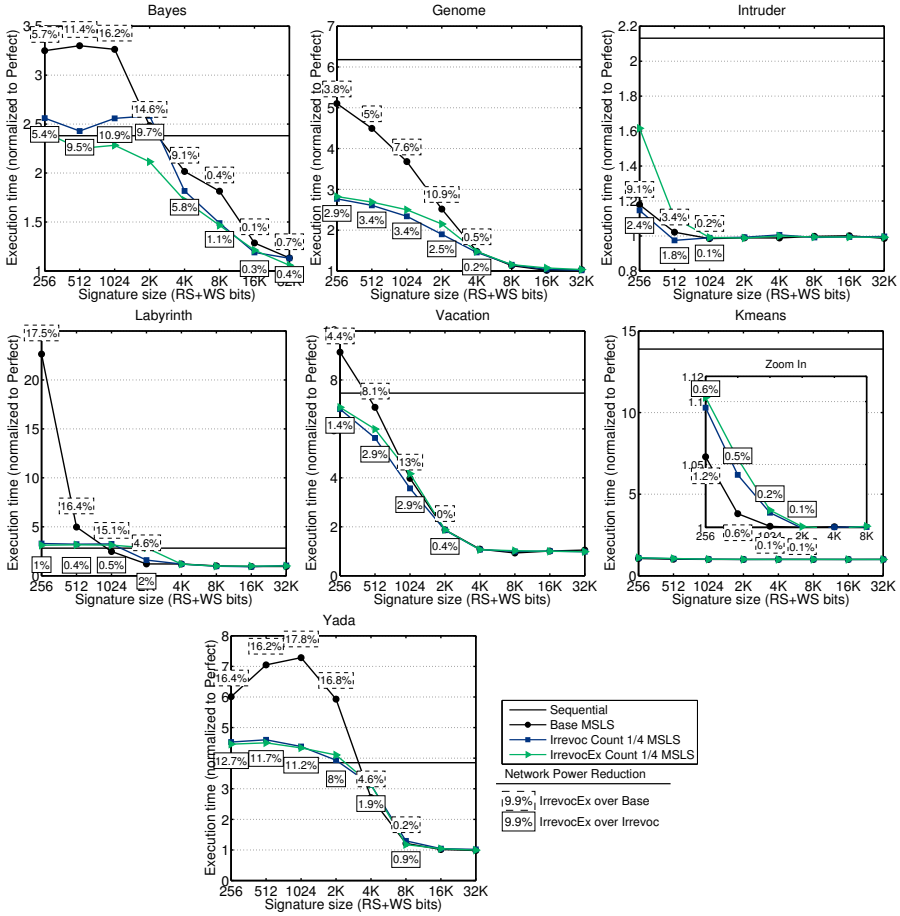

Fig. 16 Network power reduction of the irrevocability mechanisms (Count 1/4 MSLS).

cannot be aborted. Thus, transactional execution is guaranteed to progress when running both small transactions and large ones.

A novel application of irrevocability is proposed where signature occupancy is used to predict contention situations that may cause a high number of transaction aborts. This helps to save processor cycles and energy due to unnecessary aborts while assuring forward progress. An analytical model is presented that hints the benefits of irrevocability on high contention scenarios.

We implemented our proposals in the Wisconsin GEMS simulator and obtained encouraging results from the STAMP benchmark suite and a signature threshold of 1/4th the signature size. Besides execution time, network power and traffic are reduced, specially when using irrevocability with exclusion.

Acknowledgments This work has been supported by the Government of Spain, under project TIN2013-42253-P, and Junta de Andalucía, under project P12-TIC-1470.

References

1. Agarwal, N., Krishna, T., Peh, L.S., Jha, N.: GARNET: A detailed on-chip network model inside a full-system simulator. In: Int'l. Symp. on Performance Analysis of Systems and Software (ISPASS'09), pp. 33–42 (2009)
2. Alameldeen, A.R., Wood, D.A.: Variability in architectural simulations of multi-threaded workloads. In: 9th Int'l Symp. on High-Performance Computer Architecture (HPCA'03), pp. 7–18 (2003)
3. Ananian, C., Asanovic, K., Kuszmaul, B., Leiserson, C., Lie, S.: Unbounded transactional memory. In: 11th Int'l. Symp. on High-Performance Computer Architecture (HPCA'05), pp. 316–327 (2005)
4. Ansari, M., Kotselidis, C., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Advanced Concurrency Control for Transactional Memory Using Transaction Commit Rate. In: Int'l. Conf. on Parallel Processing (Euro-Par'08), pp. 719–728 (2008)
5. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM **13**(7), 422–426 (1970)
6. Blundell, C., Devietti, J., Lewis, E.C., Martin, M.M.K.: Making the Fast Case Common and the Uncommon Case Simple in Unbounded Transactional Memory. In: 34th Ann. Int'l. Symp. on Computer Architecture (ISCA'07), pp. 24–34 (2007)
7. Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing Transactional Semantics: The Subtleties of Atomicity. In: 4th Ann. Workshop on Duplicating, Deconstructing, and Debunking (WDDD'05) (2005)
8. Bobba, J., Goyal, N., Hill, M.D., Swift, M.M., Wood, D.A.: TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In: 35th Ann. Int'l. Symp. on Computer Architecture (ISCA'08), pp. 127–138 (2008)
9. Carter, J., Wegman, M.: Universal classes of hash functions (extended abstract). In: 9th Ann. Symp. on Theory of Computing, pp. 106–112 (1977)
10. Ceze, L., Tuck, J., Torrellas, J., Cascaval, C.: Bulk disambiguation of speculative threads in multiprocessors. In: 33th Ann. Int'l. Symp. on Computer Architecture (ISCA'06), pp. 227–238 (2006)
11. Choi, W., Draper, J.: Unified signatures for improving performance in transactional memory. In: Int'l. Parallel Distributed Processing Symp. (IPDPS'11), pp. 817–827 (2011)
12. Click, C.: Azul's experencies with hardware transactional memory. In: Bay Area Workshop on Transactional Memory (2009)
13. Didona, D., Felber, P., Harmanci, D., Romano, P., Schenker, J.: Identifying the optimal level of parallelism in transactional memory applications. Computing **97**(9), 939–959 (2015)
14. Goel, B., Titos, R., Negi, A., McKee, S.A., Stenstrom, P.: Performance and energy analysis of the restricted transactional memory implementation on Haswell. In: 28th Int'l Parallel and Distributed Processing Symp. (IPDPS'14) (2014)
15. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: 22nd Ann. Symp. on Principles of distributed computing (PODC '03), pp. 92–101 (2003)
16. Herlihy, M., Moss, J.: Transactional memory: Architectural support for lock-free data structures. In: 20th Ann. Int'l. Symp. on Computer Architecture (ISCA'93), pp. 289–300 (1993)
17. Hong, S., Oguntebi, T., Casper, J., Bronson, N., Kozyrakis, C., Olukotun, K.: Eigenbench: A simple exploration tool for orthogonal TM characteristics. In: Int'l. Symp. on Workload Characterization (IISWC'10), pp. 1–11 (2010)
18. Jacobi, C., Slegel, T., Greiner, D.: Transactional Memory Architecture and Implementation for IBM System z. In: 45th Ann. Int'l. Symp. on Microarchitecture (MICRO'12), pp. 25–36 (2012)

19. Kahng, A., Li, B., Peh, L.S., Samadi, K.: ORION 2.0: A fast and accurate noc power and area model for early-stage design space exploration. In: Design, Automation and Test in Europe (DATE'09), pp. 423–428 (2009)
20. Larus, J., Rajwar, R.: Transactional Memory. Morgan & Claypool Pub. (2007)
21. Machado Pereira, M., Gaudet, M., Nelson Amaral, J., Araujo, G.: Study of hardware transactional memory characteristics and serialization policies on Haswell. Parallel Computing (available online) (2015). URL http://www.sciencedirect.com/science/article/pii/S0167819115001568
22. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B., Werner, B.: Simics: A full system simulation platform. IEEE Computer **35**(2), 50–58 (2002)
23. Martin, M., Sorin, D., Beckmann, B., Marty, M., Xu, M., Alameldeen, A., Moore, K., Hill, M., Wood, D.: Multifacet's general execution-driven multiprocessor simulator GEMS toolset. ACM SIGARCH Computer Architecture News **33**(4), 92–99 (2005)
24. Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: Int'l Symp. on Workload Characterization (IISWC'08), pp. 35–46 (2008)
25. Moore, K., Bobba, J., Moravan, M., Hill, M., Wood, D.: LogTM: Log-based transactional memory. In: 12th Int'l. Symp. on High-Performance Computer Architecture (HPCA'06), pp. 254–265 (2006)
26. Mullin, J.K.: A Second Look at Bloom Filters. Communications of the ACM **26**(8), 570–571 (1983)
27. Quislant, R., Gutierrez, E., Plata, O., Zapata, E.: Hardware signature designs to deal with asymmetry in transactional data sets. IEEE Trans. on Parallel and Distributed Systems **24**(3), 506–519 (2013)
28. Quislant, R., Gutierrez, E., Plata, O., Zapata, E.L.: LS-Sig: Locality-sensitive signatures for transactional memory. IEEE Trans. on Computers **62**(2), 322–335 (2013)
29. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: 32th Ann. Int'l. Symp. on Computer Architecture (ISCA'05), pp. 494–505 (2005)
30. Reinders, J.: Intel 64 and IA-32 architectures software developer's manual, vol. 3, ch. 19. intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (2013)
31. Rughetti, D., Sanzo, P.D., Ciciani, B., Quaglia, F.: Analytical/ML Mixed Approach for Concurrency Regulation in Software Transactional Memory. In: Int'l. Symp. on Cluster, Cloud and Grid Computing, pp. 81–91 (2014)
32. Sanchez, D., Yen, L., Hill, M., Sankaralingam, K.: Implementing signatures for transactional memory. In: 40th Ann. Int'l Symp. on Microarchitecture (MICRO'07), pp. 123–133 (2007)
33. Sanzo, P.D., Re, F.D., Rughetti, D., Ciciani, B., Quaglia, F.: Regulating Concurrency in Software Transactional Memory: An Effective Model-based Approach. In: Int'l. Conf. on Self-Adaptive and Self-Organizing Systems, pp. 31–40 (2013)
34. Shriraman, A., Dwarkadas, S., Scott, M.: Flexible decoupled transactional memory support. In: 35th Ann. Int'l. Symp. on Computer Architecture (ISCA'08), pp. 139–150 (2008)
35. Sorin, D.J., Plakal, M., Condon, A.E., Hill, M.D., Martin, M.M.K., Wood, D.A.: Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. IEEE Trans. Parallel and Distributed Systems **13**(6), 556–578 (2002)
36. Spear, M.F., Silverman, M., Dalessandro, L., Michael, M.M., Scott, M.L.: Implementing and exploiting inevitability in software transactional memory. In: 37th Int'l. Conf. on Parallel Processing (ICPP'08), pp. 59–66 (2008)
37. Wang, A., Gaudet, M., Wu, P., Amaral, J.N., Ohmacht, M., Barton, C., Silvera, R., Michael, M.: Evaluation of Blue Gene/Q hardware support for transactional memories. In: 21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT'12), pp. 127–136 (2012)
38. Wang, M.D., Mihai, B., Li, L., Sharifymoghaddam, S., Steffan, G., Amza, C.: Exploring the performance and programmability design space of hardware transactional memory. In: 9th Workshop on Transactional Computing (TRANSACT'14) (2014)

39. Watson, I., Kirkham, C., Lujan, M.: A study of a transactional parallel routing algorithm. In: 16th Int'l. Conf. on Parallel Architecture and Compilation Techniques (PACT '07), pp. 388–398 (2007)
40. Welc, A., Bratin, S., Adl-Tabatabai, A.R.: Irrevocable transactions and their applications. In: 20th Symp. on Parallelism in Algorithms and Architectures (SPAA'08), pp. 285–296 (2008)
41. Yen, L., Bobba, J., Marty, M., Moore, K., Volos, H., Hill, M., Swift, M., Wood, D.: LogTM-SE: Decoupling hardware transactional memory from caches. In: 13th Int'l. Symp. on High-Performance Computer Architecture (HPCA'07), pp. 261–272 (2007)
42. Yen, L., Draper, S., Hill, M.: Notary: Hardware techniques to enhance signatures. In: 41st Ann. Int'l Symp. on Microarchitecture (MICRO'08), pp. 234–245 (2008)
43. Yoo, R.M., Hughes, C.J., Lai, K., Rajwar, R.: Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In: Int'l Conf. on High Performance Computing, Networking, Storage and Analysis (SC'13), SC '13, pp. 19:1–19:11 (2013)
44. Yu, X., He, Z., Hong, B.: An analytical model on the execution of transactional memory. In: 22nd Int'l. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD'10), pp. 175–182 (2010)