Analyzing the Sensitivity to Faults of Synchronization Primitives

Paolo Roberto Grassi
Dipartimento di Elettronica ed Informazione
Politecnico di Milano
Milano, Italy
grassi@elet.polimi.it

Mariagiovanna Sami
Dipartimento di Elettronica ed Informazione
Politecnico di Milano
Milano, Italy
sami@elet.polimi.it

Ettore Speziale *
Dipartimento di Elettronica ed Informazione
Politecnico di Milano
Milano, Italy
speziale@elet.polimi.it

Michele Tartara
Dipartimento di Elettronica ed Informazione
Politecnico di Milano
Milano, Italy
mtartara@elet.polimi.it

Abstract

Modern multi-core processors provide primitives to allow parallel programs to atomically perform selected operations. Unfortunately, the increasing number of gates in such processors leads to a higher probability of faults happening during the computation. In this paper, we perform a comparison between the robustness of such primitives with respect to faults, operating at a functional level. We focus on locks, the most widespread mechanism, and on transactional memories, one of the most promising alternatives. The results come from an extensive experimental campaign based upon simulation of the considered systems. We show that locks prove to be a more robust synchronization primitive, because their vulnerable section is smaller. On the other hand, transactional memory is more likely to yield an observable wrong behaviour in the case of a fault, and this could be used to detect and correct the error. We also show that implementing locks on top of transactional memory increases its robustness, but without getting on par with that offered by locks.

I. Introduction

The dramatic increase of multi/many-core systems' complexity leads to extensive introduction of a wide variety of parallel applications and algorithms, and therefore to the necessity of efficient and safe ways to allow synchronization among threads. *Locks* are (historically) the first and still the most popular software synchronization primitive [1]. Using locks requires the programmer to carefully avoid several common mistakes in the design of massively parallel programs, that would lead to erroneous behavior such as starvation or deadlocks; moreover, it has been argued that lock-based synchronization would make actions such as modifications or extensions of existing programs more exposed to programming errors. To avoid these risks, Herlihy and Moss [2] proposed the *Transactional Memory* concept, allowing "lock-less synchronization". Nowadays many different implementations of transactional memory have been proposed, either software [3], [4] or hardware [5], [6], [7], [8] based.

Much effort has gone into discussing various developments for locks and transactional memory, focusing on performances and - more recently - on power consumption [9], [10]. Yet, a further aspect should be taken into account as well, namely, the impact of hardware faults on the synchronization solution. The increasing integration level, density and scaling in transistor dimensions will have great impact on the reliability of next-generation systems. A non-negligible amount of *"hard"* (permanent) faults is likely to affect the chip even during its working lifetime; decreasing geometries moreover make it more probable that *"soft"* (temporary) faults will affect some of the gates during computation. In particular, "upsets" affecting memory elements should be taken care of: given a set of processors concurrently executing a task, if one of the processors hangs because of a memory fault this could block all other processors waiting for synchronization with it, so that the erroneous behavior would propagate in a dramatic fashion throughout the system.

In the present paper, we aim at analyzing the effect of faults on synchronization primitives. In particular, we compare the sensitivity to faults of locking techniques and hardware transactional memory, adopting technology-independent fault assumptions for both cases and exploring - through extensive simulations - the outcome of comparable fault distributions. Many papers present fault analysis and handling for single threaded systems [11], [12], but to the best of our knowledge, this is the first investigation about the differences between these primitives with respect to fault sensitivity performed by using experimental results.

The paper is organized as follows: Section II describes the faults we are considering, Section III presents the methodology we used for performing the simulations, Section IV shows the experimental results and Section V concludes.

II. Faults characterization

Our focus is here on soft (transient) faults, more specifically on faults identified as "Single Event Upsets" (SEUs) [13]. For our purposes, we organize faults into two different classes: the ones that affect the *general computation* - here defined *general*

* This author was supported in part by a grant from ST Microelectronics.

faults - and the ones that specifically affect mechanisms related to *critical sections*, either protected by locks or by transactions. Critical sections are particularly sensitive sections of code that are present in multi-threaded programs: wrong access to one of them by one of the concurrent threads can produce relevant errors in the program and can cause deadlocks or starvations, leading to the inability for the program to finish its execution. Hereafter we will only focus on critical section-related Single Event Upsets (SEU): we ignore other (general) types of faults, such as program counter corruption, that are beyond the scope of our analysis. Moreover, we aim at a technology-independent analysis: no assumptions are made here concerning the causes of faults, but we actually consider functional errors - affecting the outcome of specific instructions or operations. This will lead us to examine errors as affecting memory words or registers, often collapsing a number of different faults into one "equivalent" error type. In the same spirit, uniform random distributions will be adopted (thus abstracting from other possible distributions due to technological peculiarities).

According to Gawkowski et al. [14], the following outcomes can derive from applying faults to a program:

a) Correct Result (CR): the program correctly terminates its execution, computing the right value.

b) Incorrect Result (IR): the program gracefully terminates its execution, but the computed value is not correct and the system does not detect the error.

c) Fault Detected by the System (FDS): an hardware exception occurs. The system terminates the faulted program following predetermined policies.

d) Timeout (T): the program does not respect its timing requirements and is terminated by the system.

e) User-defined Message (UM): the program detects a misbehaviour, that is signalled to the user.

We follow the same classification, with the exception of User-defined Messages, since we did not add any error correction/detection machinery to the analyzed programs.

III. The Methodology Adopted

In order to obtain an indication of the respective performances of lock-based and transactional-memory-based solutions (as far as sensitivity to faults is concerned) we chose to set up an experimental environment (based on simulation tools) capable of simulating the operation of a realistic multiprocessor system as well as of supporting fault injection and simulating behavior after fault.

This choice is due to the fact that the only viable alternative would be to perform an analysis starting from the netlist of a hardware device. This device should support both lock based and transactional synchronization primitives. Moreover, it should be a neutral, publicly available benchmark (a personal choice would risk to be biased). Since such a device was not available, we decided to go for a simulation approach, so as to provide at least a first analysis that, although less precise, is more general and a good starting point for further work.

To obtain the experimental results presented here, we started from the SESC simulator. SESC is "a microprocessor architectural simulator [...] that models [...] chip multiprocessors, [...]". CPUs used as nodes are MIPS processors, with "a full out-of order pipeline with branch prediction, caches, buses, and every other component of a modern processor necessary for accurate simulation" [15]. More specifically, SESC operates at at functional-block level simulating the execution of a program.

In order to support the simulation of parallel programs, SESC provides its own implementation of a POSIX-like threading library, called *libapp*. libapp is much simpler than pthread, but it provides all that is needed for the aim of the present paper - at least insofar as lock-based synchronization is concerned. Namely, libapp provides fork/wait primitives and lock/unlock primitives. While this allows us to proceed with the analysis of fault impact on lock-based solutions, to perform our comparison we also need an implementation of a transactional memory - which is not provided by SESC.

On the other hand, SuperTrans [16], developed by University of Florida's Advanced Computing and Information Systems Laboratory, is "a multicore, cycle-accurate and multiple issue simulator built on top of the SuperESCalar (SESC) framework that is capable of simulating three of the most common dimensions of hardware transactional memory (Eager/Eager [7], [8], Eager/Lazy [7], [17], Lazy/Lazy [6])". SuperTrans, just as SESC, is released as open source. It includes all that is part of SESC (therefore, the lock based management of memory) plus a transactional memory module. For these reasons, we chose SuperTrans as the tool for transactional-memory related simulations; being based on SESC, it granted the kind of consistency that was mandatory to compare results of simulations obtained on the two systems.

In order to explore the effects of faults, we modified SuperTrans by adding a new software component, that we named *fault injector*, allowing us to specify where and when to inject faults during the simulation, so that we can observe the outcome of the management of the mutual exclusion between two or more processes trying to access a single critical section. The fault injector can support an arbitrary number of faults. The characteristics of the faults can be completely specified by the user or randomly generated.

IV. Impact of Faults on Synchronization Mechanisms

In order to evaluate how faults affect the behavior of programs run by systems that use, respectively, locks or transactional memory to protect the critical sections, we carried out an extensive experimental campaign, using a small set of synthetic

Algorithm: $shared\_inc\_lock$

Data: a shared counter cnt
Result: cnt safely incremented by 1

```
1 lock_acquire(cnt.lock)
2 cnt.n ← cnt.n + 1
3 lock_release(cnt.lock)
```

Figure 1: Shared counter update. Locking functions guarantee *mutual exclusion* between threads while concurrently incrementing the counter

Algorithm: $shared\_inc\_trans$

Data: a shared counter cnt
Result: cnt safely incremented by 1

```
1 trans_begin()
2 cnt ← cnt + 1
3 trans_commit()
```

Figure 2: Shared counter update exploiting transactional memory. If a conflict is detected during a transaction, it is aborted and restarted by the hardware

Algorithm: $lock\_acquire$

Data: a lock $lock$
Result: $lock$ locked by current thread

```
1 while XCHG(lock, LOCKED) = LOCKED do
2     repeat NOP until lock ≠ LOCKED
3 end
```

Figure 3: Implementation of $lock\_acquire$

Algorithm: $lock\_release$

Data: a lock $lock$
Result: $lock$ unlocked

```
1 lock ← UNLOCKED
```

Figure 4: Implementation of $lock\_release$

benchmarks (depicted in Table I) that implement well known concurrency problems, such as *shared counter* or *reader/writer interactions* [18]. Using such simple examples allows us to easily inject faults exactly in the registers and cache lines that will be accessed by the code while inside a critical section. Therefore we can verify the effect of faults on the more likely sources of problems related specifically to the synchronization mechanism adopted rather than to the general effects of faults on program's execution. Moreover, these small benchmarks share the same structure of most complex concurrent applications, so that the results we obtain are actually general. Studying the effect of faults on synchronization primitives has a direct impact on determining how the behaviour of the application will change because of them. In fact, many years of research on operating systems [18] prove the importance of the correct behaviour of such primitives.

We will now describe in detail how faults are injected in the micro-benchmarks and what the results obtained using the *SC* micro-benchmark as a running example. Section IV-A reports on fault injection in lock-based critical sections, while Section IV-B refers to transactional-memory-based critical sections, Section IV-C describes faulting critical sections protected with transactional-memory-based locks - a solution that, while non-realistic, allows completing our fault-related analysis with this alternative derived from the two basic criteria. Finally, Section IV-D presents the experimental campaign setup and its results.

A. Lock-based Critical Sections

From the users perspective, protecting a critical section cs requires invoking a $lock\_acquire$ function before entering cs. This guarantees that no more than one thread at a time enters the critical section. To leave cs, a thread must invoke a $lock\_release$ function. This allows other threads to access cs. Figure 1 shows how these routines can be employed to safely increment a shared-counter.

Such locking/unlocking routines are built on top of hardware synchronization instructions, such as *atomic eXCHanGe*, *Compare And Swap*, and *Load Linked/Store Conditional*. No other *ad hoc* hardware capabilities are exploited to implement the routines: the remaining code segments are implemented using standard instructions. Figures 3 and 4 show $lock\_acquire$ and $lock\_release$ routines respectively.

Any fault generated inside a critical section can corrupt the current thread's private data, as well as the private data of other threads and shared data. This happens because a critical section's body contains only non lock-related instructions and the locking algorithm has no knowledge of the data accessed and of the instructions executed inside it.

If we consider critical section boundaries, identified by $lock\_acquire$ and $lock\_release$ routines, we see that a fault affecting data accessed by these routines is catastrophic because they control the *access to the critical section*. Even in the presence of transient faults, the program behavior is radically modified: more than one thread will access the critical section at the same time, performing a computation at the *wrong time*. The faulted program behavior matches classical concurrent programming errors, such as *lost update*, *dirty read/write*, ...

For our experimental campaign, we start by injecting faults affecting $lock\_acquire$. The most important operation performed here is $XCHG$ (Figure 3, Line 1): it atomically replaces the memory word where $lock$ resides with the $LOCKED$ constant,

returning the value stored there before the swap took place. We identify three elements such that faults affecting them are critical for the synchronization process, namely: $lock$, the register containing the $LOCKED$ constant, and the return value.

To emulate faults on $lock$ we consider them just by their outcome: having the program reading/writing the wrong memory location, therefore causing $XCHG$ to return a wrong value. Such value is later read (Figure 3, Line 1) by a comparison instruction to detect whether to enter the critical section, so this fault can allow the current thread to enter the critical section, even if the lock is not held. The program behavior cannot be predicted, and both CR and IR can be observed. A write on the wrong address could be detected, depending on the specific address, if a FDS situation (e.g. segmentation fault) occours.

Altering the $LOCKED$ word results in writing the wrong marker in the $lock$ memory location. If it turns out to be equal to the $UNLOCKED$ marker, the current thread enters the critical section without the other threads being aware that the lock has been taken. Therefore, they can enter the section too, leading to wrong behaviour. We can observe the same behaviour also if the written marker is invalid, because every value not equal to $LOCKED$ allows entering the critical section. We can observe CR if the dynamic schedule does not result in a data race, IR or FDS otherwise.

A transient fault on the return value can result in two different behaviours: if the faulted return value is equal to $LOCKED$, the current thread spends some cycles (Figure 3, Lines 1 and 2) waiting for the lock to be released, without corrupting data. Otherwise, the current thread enters the critical section, incurring into a potential data race. We can observe the same program behaviour as in the previous case: CR, IR, or FDS.

As a final remark, it is worth noting that in the case of a thread trying to enter a critical section it is very unlikely to incur into T behaviour (provided only transient faults are applied, as in our experiments). For this to happen, the value accessed through the $lock$ variable (Figure 3, Line 1) should always be equal to the value of the $LOCKED$ constant: this requires either to continuously fault $lock$ in such a way to end up reading from memory locations containing the $LOCKED$ value, or to fault the return value of the $XCHG$ instruction every time in such a way that it results equal to $LOCKED$. Similar considerations apply to the spin wait loop, too (Line 2).

The $lock\_release$ routine is a simple store to memory. Its behaviour can be altered by injecting faults on $lock$ and on the $UNLOCKED$ marker. Modifying $lock$ shows the same behaviour as writing to an invalid memory address, potentially generating CR, IR, and FDS behaviours.

Finally, a fault affecting $UNLOCKED$ results in generating an invalid marker that corrupts $lock$, but the locking algorithm is not influenced: the first thread entering into the critical section restores $lock$ to a consistent state. On the other hand, writing the valid but incorrect value $LOCKED$ results on T behaviour: the lock is released incorrectly, preventing any thread from entering the critical section.

B. Transactional Memory-based Critical Sections

In order to protect a critical section using transactional memory, the user employs three routines: $trans\_begin$ (instructing the transactional memory to save the current context), $trans\_commit$ (to publish the memory operations performed), and $trans\_abort$ (to explicitly terminate and restart a transaction). Figure 2 shows how transactional memory can be used to protect a shared counter update.

In transactional memory approach, critical section access control is distributed; every memory operation inside a critical section is validated by the transactional controller in order to detect conflicts. Detection is performed by analyzing the *read set*, (the set of memory locations read by a thread), and the *write set*, (the set of memory locations written by a thread). To emulate errors corrupting the *read set* as well as the *write set*, requires one can collapse the various fault causes into faults affecting the addresses manipulated by the transactional controller. Therefore, we will inject faults near memory access opcodes so as to affect the system immediately before memory access.

Corrupting the read set can be modeled as reading from the wrong memory location. A transactional load, LWX, both interacts with the transactional controller and fetches data from memory. As a result, the read set of the faulted processor becomes inconsistent and a wrong value is read from the memory. If the wrong value is used for subsequent computations, it can produce IRs. The same behaviour can occur even if the read value does not directly produce a corrupted value. In fact, the transactional controller could be unable to detect a conflict due to the corrupted read set, thus allowing a transaction to commit when it should have been aborted instead. Moreover, if the corrupted address is later used for a memory store to a location not accessible by the faulted processors, a FDS occurs.

If faults lead to corrupting the write set, the same behaviour can observed. In this case the faulted instruction is the transactional write, SWX; as in the case of LWX, the instruction also interacts both with the transactional controller and the memory. Depending on the fault-affected value, an IR or FDS can occur. The difference with respect to faulting the read set is that an FDS can occur immediately.

Hardware implementations of transactional memory introduce three new opcodes, namely $XBEGIN$, $XCOMMIT$, and $XABORT$ respectively implementing $trans\_begin$, $trans\_commit$, and $trans\_abort$. All these operations do not use general purpose hardware; they interact directly with the transactional memory controller, thus to simulate faults relative to them we cannot just inject faults into registers or non-transactional memory, but we have to fault the simulated hardware primitives

Algorithm: $trans\_xchg$

Data: an address $addr$
Data: a value val
Result: val written into $addr$, old value returned

```
1 trans_begin()
2 old ← mem[addr]
3 mem[addr] ← val
4 trans_commit()
5 return old
```

Figure 5: Atomic exchange implemented using transactional memory. It is used as a building block for transactional memory-based locks

```
[upReg]
generator = 'uniform'
regNo = 'R18'
kind  = 'bitFlip'
atTime = 1100
```

Figure 6: An example of fault taken from the configuration file. A bit-flip fault named `upReg` will be applied to register `R18` at 1100^{th} cycle of the simulation

themselves. Faults concerning this scenario corrupt processor context saved by $XBEGIN$ and restored by $XCOMMIT$ and $XABORT$. Since these faults would be very much dependent on a specific implementation and technology, we do not consider them; obviously, extending the set of faults would increase the sensitivity to faults of the system.

C. Transactional Locking-based Critical Sections

As shown in Section IV-A, we can inject a wide variety of faults on locks, but the lock is directly manipulated only at critical section bounds, so there is not much possibility for such faults to happen. Every other fault happening inside a critical section protected by locks is not related to locks themselves: as such it could happen whatever the synchronization primitive being used, and is therefore not interesting for this study. On the other hand, transactional memory is vulnerable to a narrow class of faults, see Section IV-B, but they expose more faulting opportunity because as long as the transaction is active every memory access could be influenced by faults in the *read set* or the *write set*.

This observation led us to try to implement locks "on top of" transactional constructs. While this is not a viable solution for real systems, it allows us to study whether transactional memory helps reducing faulting opportunities. The locking and unlocking algorithms are the same used for lock-based critical sections (Figure 3 and 4). In order to exploit transactional memory, we replaced the $XCHG$ instruction with an equivalent routine written using transactional constructs. Its implementation can be seen in Figure 5. The lock release routine has been modified so as to be protected by a transaction.

For faults happening inside the critical section we can make the same observations as for locks, because the critical section does not contains any special instruction.

On the other hand, we note that injecting faults on critical section boundaries requires injecting faults on the transactions protecting the atomic exchange. The kind of faults that can be injected are the same as for transactional memory: basically, we can fault the read set and the write set.

In this particular critical section, the read set and the write set are identical: they consist just of the word used to store the lock. Faulting the lock address can thus produce FDS, IR, or T behaviours. The first arises when the faulted lock address refers to a memory region that cannot be written by the faulted thread. If the word identified by the faulted address can be written, a data race can occur, possibly generating either CR or IR. The T behaviour occurs when reading from the faulted address causes the faulted thread to spend too much time in the lock busy-wait loop.

D. Results of the Experimental Campaign

Our experimental campaign focused on the micro-benchmarks reported in Table I. We coded each micro-benchmark in three different flavour, each employing a different primitive to protect its critical sections. The *lock* flavour, uses locks, *trans* uses transactions, while *trans-lock* adopts locks implemented by means of transactions.

Each micro-benchmark was first run without applying any faults. Observing the execution trace we detected points where faults could be injected, as suggested in Section IV-A, IV-B, and IV-C. Each flavour exposes different faultable points. Faults will affect execution with the *lock* flavour while acquiring and releasing the lock. The *trans* flavour is faultable while accessing the read set and the write set, i.e. near each LWX and SWX. The *trans-lock* flavour exposes the same faultable points as *trans*, but the critical section is shorter.

We wanted to see the evolution of the behaviour of the micro-benchmarks subject to an increasing number of faults. Therefore, for each of them we injected an increasing number of faults, from 1 to 4. For each benchmark, for each given number of faults, we performed 960 [1] runs. Before each run we randomly extract i faultable points taken from those observed by analyzing

[1] 240 for RWL-trans.

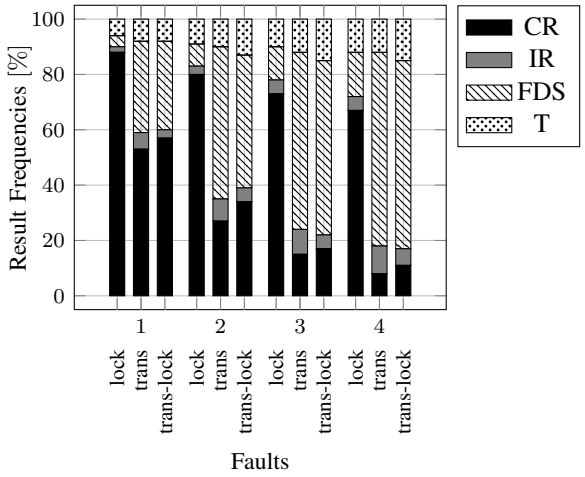

Figure 7: Distribution of benchmark results, varying the number of applied faults

Table I: Benchmarks

SC	concurrent increment of a shared counter. Each thread performs 8 atomic increments.
SMC	concurrent increment of shared counters. Each thread executes 4 critical sections, incrementing 16 counters each time.
RW	reader/writer problem. Threads are partitioned into two equally sized sets: readers and writers. Writers produce items writing them into a global buffer. Readers read items from the buffer. When all items have been produced, the readers concurrently write all the read items into another buffer read by the main thread to perform a final sanity check. Buffers are implemented using arrays.
RWL	reader/writer problem. Same behaviour of *RW*, but shared buffers are implemented using single-linked lists.

Table II: Benchmark results. For each configuration, 960 runs have been performed (240 for RWL-trans)

Benchmark		**CR** [%]				**IR** [%]				**FDS** [%]				**T** [%]			
		$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$	$\mathbf{F_1}$	$\mathbf{F_2}$	$\mathbf{F_3}$	$\mathbf{F_4}$
SC	lock	82	76	71	63	3	5	6	9	4	9	13	17	11	10	10	11
	trans	50	25	13	7	8	12	14	14	42	63	73	79	0	0	0	0
	trans-lock	55	35	18	10	5	6	8	8	40	59	74	82	0	0	0	0
SMC	lock	92	86	81	80	4	4	6	5	3	8	9	11	1	2	4	4
	trans	51	24	14	10	8	10	12	14	40	64	73	74	1	2	1	2
	trans-lock	57	34	21	13	6	10	10	11	37	55	69	75	0	1	0	1
RW	lock	87	77	68	61	3	3	4	5	4	9	15	19	6	11	13	15
	trans	59	33	18	9	3	4	4	5	19	38	46	56	19	25	32	30
	trans-lock	58	30	12	11	1	2	7	2	25	38	54	59	16	30	27	28
RWL	lock	89	83	73	64	0	1	2	3	4	8	10	16	7	8	15	17
	trans	48	19	11	4	3	2	3	3	27	48	60	64	22	31	26	29
	trans-lock	56	35	17	10	1	2	2	5	26	39	54	58	17	24	27	27

the execution trace. To allow for some randomness, each fault was randomly applied between 1 and 4 cycles before the time instant it was registered in the original execution trace. In case of faults applied to registers, we randomly generated the number of the register bit to fault. For faults applied to cache line reads, we randomly generated the loaded word bit to fault. Figure 6 shows a generated fault entry in the SESC configuration file format.

Table II reports individual benchmark results, while Figure 7 shows the percentage of *CR*, *IR*, *FDS*, and *T* for each flavour, varying the number of applied faults.

The *lock* flavour is the most robust, because there are fewer points where a fault can be injected. Moreover, the fault must be injected at a precise time, otherwise the locking algorithm tends to mask the fault and thus overcomes a previous soft fault. In fact, the locking algorithm usually rewrites the content of the lock word at the beginning of the critical section, while trying to acquire the ownership via the $XCHG$ instruction, and at its end, while releasing the lock. Moreover, not all faults injected on locks can be observed, because even if two threads happen to enter in a critical section at the same time, they could not incurr in a data race, depending on the specific scheduling taking place.

Looking at Table I we see that the *trans* flavour obtains the worst outcome, with less CR compared to the *lock* flavour, because transactional memory exposes more faultable points. However, the probability that a fault will be detected (FDS) is greater, because most failures are due to accesses to wrong memory areas. These are detected by the operating system and could, in principle, be used to perform error correction, thus increasing the number of correct results.

Finally, implementing lock on top of transactional memory, i.e. the *trans-lock* flavour, increases the robustness with respect to *trans*, because each transaction lasts only as long as needed to change the lock value, but it cannot achieve the robustness of the *lock* flavour, because as short as that time span can be, every single access to memory during it can be subject to faults. Let us now analyse in detail the outcome of each benchmark.

f) Shared Counter and Shared Multi Counter: the critical section associated to *SC* is the shorter of all the benchmark suite, while *SMC* employs a longer critical section, updating more than a shared counter at time.

The *lock* flavour is the most susceptible to short critical sections. Indeed, on such scenario the program hot spot is *lock acquisition*, so any fault that induces spending some extra cycles in the lock waiting loop, greatly lowers performance, generating a considerable amount of T behaviour. When increasing the length of the critical section, the number of T behavior decreases, as shown by the *SMC* micro-benchmark, where we can observe a greater number of CR.

Both *trans* and *trans-lock* flavours follow the same trends in both *SC* and *SMC*. To obtain T behaviour, read and/or write sets of a transaction must be faulted in order to read/write data from/to the shared data, forcing an abort. Both the micro-benchmarks have a small amount of shared data, so the probability of this outcome is negligible.

g) Reader/Writer and Reader/Writer List: the *RW* micro-benchmark uses arrays to implement shared buffers, while *RWL* relies on single-linked list, thus critical sections are longer and access memory more frequently.

Locking-based techniques exhibit the same behaviour in the two micro-benchmarks. On the other hand, the *trans* flavour is heavily influenced by using single-linked lists. Using more complex structures results in more memory accesses, mostly related to list navigation. Thus, the probability of incurring into a *FDS* increases.

V. Concluding Remarks

In this paper we analyzed the behavior of locks and transactional memory when they are affected by faults. We injected from 1 up to 4 faults during the execution of selected benchmarks and analyzed the outcome of the execution. As it is easy to understand, while the number of faults grows, the probability of a visible failure increases. The important result is that locks proved to be more fault resilient because they expose a smaller "faultable surface", and it is therefore more unlikely for a fault to have the execution fail. We did not consider a specific hardware implementation, and focused only on observing the functionality of the synchronization primitives under the effect of faults. Nonetheless, it should be considered that transactional memory requires specialized hardware components to be added to the system, and this components are themselves subject to faults. This suggests that the actual fault tolerance of transactional memory could be lower than our results suggest. Further experimental campaign should be conducted in order to prove this point. On the other hand, our results show that with transactional memory the system is more likely to be able to detect the presence of faults. Further experiments could determine whether fault detection and recovery capabilities could be more effective or easier to implement in a transactional memory based system.

References

[1] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[2] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *ISCA*, 1993, pp. 289–300.

[3] N. Shavit and D. Touitou, "Software Transactional Memory," in *PODC*, 1995, pp. 204–213.

[4] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III, "Software Transactional Memory for Dynamic-sized Data Structures," in *PODC*, 2003, pp. 92–101.

[5] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun, "A Scalable, Non-blocking Approach to Transactional Memory," in *HPCA*. IEEE Computer Society, 2007, pp. 97–108.

[6] L. Hammond, V. Wong, M. K. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional Memory Coherence and Consistency," in *ISCA*. IEEE Computer Society, 2004, pp. 102–113.

[7] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded Transactional Memory," in *HPCA*. IEEE Computer Society, 2005, pp. 316–327.

[8] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "LogTM: Log-based Transactional Memory," in *HPCA*. IEEE Computer Society, 2006, pp. 254–265.

[9] T. Moreshet, R. I. Bahar, and M. Herlihy, "Energy Reduction in Multiprocessor Systems Using Transactional Memory," in *ISLPED*, K. Roy and V. Tiwari, Eds. ACM, 2005, pp. 331–334.

[10] E. Gaona-Ramírez, R. Titos-Gil, J. Fernández, and M. E. Acacio, "Characterizing Energy Consumption in Hardware Transactional Memory Systems," in *SBAC-PAD*, 2010.

[11] C. Wang, H.-S. Kim, Y. Wu, and V. Ying, "Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection," in *CGO*. IEEE Computer Society, 2007, pp. 244–258.

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *CGO*. IEEE Computer Society, 2005, pp. 243–254.

[13] C. Constantinescu, "Trends and Challenges in VLSI Circuit Reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[14] P. Gawkowski, J. Sosnowski, and B. Radko, "Analyzing the Effectiveness of Fault Hardening Procedures," in *IOLTS*. IEEE Computer Society, 2005, pp. 14–19.

[15] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC Simulator," 2011. [Online]. Available: http://sesc.sourceforge.net/

[16] J. Poe, C.-B. Cho, and T. Li, "Using Analytical Models to Efficiently Explore Hardware Transactional Memory and Multi-Core Co-Design," in *SBAC-PAD*. IEEE Computer Society, 2008, pp. 159–166.

[17] R. Rajwar, M. Herlihy, and K. K. Lai, "Virtualizing Transactional Memory," in *ISCA*. IEEE Computer Society, 2005, pp. 494–505.

[18] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation*, 3rd ed. Prentice Hall, 2006.