On Making Transactional Applications Resilient to Data Corruption Faults

Mohamed Mohamedin
Virginia Tech
mohamedin@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

***Abstract*—Multicore architectures are becoming increasingly prone to transient faults and data corruption. Relying on a multicore architecture is the common solution for increasing performance and scalability of core applications including transactional applications. In this paper we present SoftX, a low-invasive protocol for supporting execution of transactional applications relying on speculative processing and dedicated committer threads. Upon starting a transaction, SoftX forks a number of threads running the same transaction independently. The commit phase is handled by dedicated threads for optimizing synchronization's overhead. We conduct an evaluation study showing the performance obtained with the implementation of SoftX on a 48 cores AMD machine, running List, Bank and TPC-C benchmarks. Results reveal better performance than classical replication-based fault-tolerant systems and limited overhead with respect to non fault-tolerant protocols. We ported SoftX to a message-passing architecture, Tilera TILE-Gx. Hardware message-passing is an important emerging trend in multicore architectures. Our experiments on Tilera show that SoftX is still more efficient than replication.**

***Index Terms*—Multicore processing; Fault tolerance; Transactional systems**

I. Introduction

Multicore architectures are the current reality for improving parallelism, scalability and performance of applications. In particular, this is the case of transactional applications that are inherently parallel due to the abstraction of transaction. A single thread application performs transactions that are executed concurrently with others invoked by parallel application threads.

When those applications need to be resilient to faults, then a fault-tolerance protocol is needed for managing the concurrency among transactions, replicating objects on multiple sites or writing them on multiple storage systems. Both the solutions implicate additional costs for acquiring a distributed infrastructure but, beyond this, they have a negative impact on application performance because they introduce additional overhead such as network communication, distributed synchronization and interaction with stable storage, that can degrade overall performance even by orders of magnitude.

Faults can be roughly classified in transient and permanent. Transient faults can cause data corruption. They have many sources due to software bugs and hardware errors. In addition, some software bugs are difficult to detect and reproduce like race condition bugs, which only appear in some schedules under certain conditions. Soft-errors [1] belong to the category of hardware-related errors and they are very difficult to detect or expect/predict. Specifically, soft-errors may happen anytime during application execution. They are caused by (external) physical phenomena [2], e.g., cosmic particle strikes, electric noise, which cannot be directly managed or predicted by application designers or administrators. As a result, a soft error is silent, the hardware is not affected by interruption, but applications may crash or behave incorrectly.

This kind of faults are nowadays becoming a concrete problem to face with because of the proliferation of multicore architectures. In fact, the trend of building smaller devices with increasing number of transistors on the same chip is allowing designers to assemble these powerful computing architectures. However, although the soft error rate of a single transistor has been almost stable over the last years, the error rate is now growing due to rapidly increasing core counts [2]. A soft error can cause a single bit in a CPU register to flip (i.e., residual charge inverting the state of a transistor). Most of the time, such an event is likely to be unnoticed by applications because they do not use that value (e.g., unused register). However, sometimes, the register can contain an instruction pointer or a memory pointer. In those cases, the application behavior can be unexpected. In this paper we focus on soft-errors as a good representative of transient faults that are random, hard to detect, and can corrupt data.

Widely used replication solutions [3], [4], [5] are usually more suited for permanent failures, where the entire node goes down and it can stop or restart according to the adopted failure model, than transient faults. Byzantine behaviors are usually connected with transient faults because, when an application has a transient fault, it can arbitrarily misbehave. However, byzantine solutions (such as [6], [7]) are more general because they are designed for minimizing assumptions on execution's behavior and for malicious behavior in general. As a result, their impact on system's performance could be much higher than what is actually needed for solving the problem of transient faults that, indeed, can be seen as a small part of the bigger picture of byzantine fault-tolerance. As an example, our solution does not target any malicious application behavior or security hazard.

A famous example for data corruption effect in production systems is the major outage of Amazon S3 service[1]. In this

[1] http://status.aws.amazon.com/s3-20080720.html

incident, a single-bit flip corrupted a message and the effect propagated from one server to another. They managed to fix it by taking the system down, clearing the system state, and restoring the system back. This part of the system was not protected against data corruption errors. The downtime was over seven hours and initiated by a single transient fault.

We tackle the challenge of solving transient faults by building a low-intrusive software infrastructure that guarantees reasonable good performance, even though worse than the original, non resilient to transient faults, version of the application, but better than typical replication-based solutions. Our proposed solution guarantees both safety and liveness. The targeted system should always return correct results without any downtime.

In this paper we present *SoftX*, our framework for making a transactional application resilient to transient faults. SoftX's main goal is preventing data corruption caused by a transient fault from propagating in main memory, without a significant impact on application's performance (i.e., 10× worse). This goal is reached by SoftX leveraging on speculative processing [8], [9], [10]. When a transaction is activated, SoftX captures the transaction and forks a number of threads that replicate the execution of the transaction in parallel, speculatively on different cores of the system. We assume an underlying hardware architecture equipped with an enough core count such that the speculative execution does not hamper the application's progress. The outcome of each thread is then compared with the others using dedicated *committer threads*. If it corresponds to the majority outcome, then the transaction can commit and the control is returned to the application. This approach masks transient faults happened during the execution on some core with the speculative execution on other cores.

A straightforward solution to transient faults problem that supports both safety and liveness is state-machine replication (SMR) [11]. But SMR approach has several sources of overhead. Request must be wrapped in a network message, totally ordered, executed in-order, and finally uses voting to compare results and accepts the majority result. SMR requires also a high communication bandwidth. In order to map SMR approach which is designed for distributed system to a centralized multicore machine, computational resources, as well as memory, should be partitioned into replicas [12].

In SoftX, computational resources and memory are not partitioned and no ordering protocol is required before to process transactions (e.g., [10], [8]) or after for validating transactions (e.g., [13]). Moreover, hardware cores are reused by other transactions after a commit event; they are not preassigned to certain application threads such that the overall system's load is more balanced.

As a practical design choice, SoftX follows the same abstraction as Software Transactional Memory (STM) [14] where transactions are simply enclosed in atomic blocks and the burden of managing concurrency and guaranteeing isolation and atomicity is on the STM protocol. This decision does not represent a limitation for our framework. It can be easily integrated with other transaction abstractions. We decided to use the STM's because this framework is appealing and transparent from the programmer standpoint.

In order to assess the performance of SoftX, we implemented it on top of the Rochester STM (RSTM) library [15] a famous framework for developing and running STM algorithms. As competitors, we selected: a well known STM system such as NOrec [16] as representative of protocols that are not resilient to transient faults; a modified version, for being resilient to transient faults, of the classical SMR approach [12], where transactions are ordered before their execution; and PBFT [6] as a representative of Byzantine fault-tolerant systems. This way we attempt to provide an overview about SoftX's cost and its performance against fault-tolerant, as well as non fault tolerant approaches.

Results on a 48-cores AMD server, using well-known transactional benchmark such as List, Bank and TPC-C [17], reveal that SoftX has an overhead with respect to NOrec that is limited to 1.1× except for Bank which is characterized by very small transactions. Thus, synchronization overhead is more evident. Real applications usually have larger transactions. Compared to other fault-tolerant approaches, SoftX gains against the SMR approach by an average of 1.6× and gains against PBFT by an average of 4.5×.

Hardware message-passing communication is an important emerging trend in multicore architectures. Given that SoftX targets multicore architectures, we believe it is important to study our proposal on both: current shared-bus architectures, as well as emerging message-passing architectures. In order to reach this goal, we ported SoftX to the Tilera TILE-Gx [18] board, a message-passing architecture. Message-passing architectures do not suffer from the limited bandwidth of the shared-bus architectures. Results show that SoftX still outperforms replication-based approaches. SoftX's overhead compared to non-fault-tolerant approach (i.e., NOrec) is reduced to about 40% on average. Replication-based approaches also take advantage of the hardware message-passing network. Their results are now closer to SoftX.

SoftX is optimized for transactional applications because we believe they fulfill a key role in the space of transient faults especially because their nature does not allow any kind of unexpected and unmanageable data corruption.

To the best of our knowledge, this paper is the first attempt to reduce the scope of data corruption to transactional processing without the overhead of classical replication-based solutions. We propose a solution that reduces overhead and increases resources utilization while keeping safety and liveness.

The rest of the paper is organized as follows: we provide a comprehensive taxonomy of transient fault-tolerant techniques in Section II. In Section III, we discuss the categories of transient faults that SoftX can tolerate. Sections IV discusses how SoftX is designed and implemented in details. SoftX evaluation and experimental results are in Section V. We conclude the paper in Section VI.

SoftX's full implementation, with sources and test-scripts, is available at www.hyflow.org/downloads/softx.zip.

II. Related Work

Many techniques are proposed to handle transient faults that cause data corruption or *silent* data corruption. All techniques are targeting safety where correct results are the main target. But not all of them guarantee liveness.

Checkpointing is a well known technique in High Performance Computing (HPC). The system state is checkpointed periodically and fault detection mechanisms are installed to detect data corruption and restore the system to a correct system state. Other approaches in HPC rely on more complex algorithms. They are known as fault-tolerant algorithms. Some scientific problems can be modified to support error detection and correction using some encoding mechanisms like checksums. These approaches are algorithm specific and they are difficult to generalize. Encoded processing incurs high processing overhead and requires extra memory. In [19] more information about such techniques can be found.

Some less efficient error detection techniques, in terms of error coverage, are bases on assertions/invariants or symptoms [20]. Assertions are conditions that must exist before and after program state changes. Symptoms refer to program activities that are usually related to faults. For example, executing an unknown instruction, and misprediction of a high confidence branch. These techniques do not provide full error coverage and require another mechanism for recovery like checkpointing.

Hardening crash-tolerant systems (e.g., [21]) can tolerate transient faults by using error detection mechanisms. When an error is detected, it is converted to a crash fault by taking the node down. Hardening can be applied to replication-based fault tolerant systems that tolerate crash faults. In this case, it supports both safety and liveness, but it depends on the accuracy of the error detection. Also, error detection must detect faults quickly before they can propagate.

Byzantine fault tolerant (BFT) systems can tolerate transient faults by default. When a transient fault occurs, the program can continue its execution (i.e., without crash) while producing incorrect behaviors. But BFT systems have a broader scope as they also target malicious activities (e.g., intrusion). As a result, usually their overhead is high. Most BFT systems are distributed systems and rely on state-machine replication. In state-machine replication, requests must be totally ordered before execution to guarantee reaching the same state on each replica independently. Each replica executes requests in the same order. Reaching order agreement requires exchanging multiple messages. Some BFT systems execute requests sequentially to guarantee determinism [6], [22]. Other BFT systems execute independent requests in parallel to increase system throughput [23], [24], [25]. In [7], a centralized system is split using isolated virtual machines which represent typical BFT replicas. SoftX aims for reducing the overhead of the fault-tolerant protocol without partitioning the machine resources between different replicas and without replicating system memory.

Hardware fault-tolerant architectures are more expensive and limited in the number of faults they can tolerate due to the cost of the redundant hardware. For example, *NonStop* advanced architecture [26] uses two or three replicas. Each replica executes the same instructions stream independently and a voter compares outputs. Hardware resources (memory, disks) are split between replicas and isolated from each other.

Compared to the above techniques, SoftX is a pure software solution and it inherits both the checkpointing and replication advantages. Using transactional memory to speculatively execute a transaction is similar to taking a checkpoint at the beginning of each transaction. If an error is detected, the execution returns to that checkpoint and retry. SoftX uses also the same principle of replication by executing the same transaction multiple times in parallel. However, SoftX's speculative execution is more lightweight: data are not replicated and there is no fixed number of replicas. Thus, synchronization overhead is minimal and cores can be reused by other threads.

SoftX also supports parallel execution of transactions where multiple thread groups can execute together in the same time. Transactional memory mechanisms coordinate access to shared data. Using dedicated committer threads reduces the synchronization overhead more by minimizing the number of CAS operations. Instead of having all threads competing on the same shared lock, each thread communicates directly with committer threads via a private variable.

Works tackling similar problems (i.e., redundant parallel execution) of SoftX has been presented in [27], [28]. In [27], authors use relaxed determinism which is similar to relaxed memory consistency model of modern processors. However, developer must insert determinism hints when precise ordering is needed. SoftX is transparent to the developer. It uses transactions, which define exactly where the parallel speculated execution starts and ends. Moreover, SoftX is implemented as part of an STM library and requires no change to the operating system or compiler.

Transactional applications are increasingly using Software Transactional Memory (STM) algorithms [14], [16], [29], [30] to overcome the programmability and performance challenges of in-memory processing on multicore architectures.

A locking mechanism based on the idea of executing lock-based critical sections in remote threads has been recently explored in [31], [32]. SoftX exploits a similar approach using dedicated committer threads for processing transactions' commit phases.

III. Assumptions and Applicability

SoftX does not replicate data in memory therefore, when a hardware error that affects memory (e.g., soft errors) occurs, it relies on the detection and correction techniques, like ECC (Error-Correcting Code), to avoid propagation of such a error into registers. SoftX protects data by allowing only committer threads to write in memory. Speculative threads (groups) have read-only access to shared data. Writes are buffered and sent to committer threads to decide if they can be committed safely or a conflict/error happened. Moreover, each committer thread keeps an undo log of its writes to shared data so that writes

can be validated and if a corruption is detected, memory can be restored using the undo log.

SoftX works on a single machine, thus it cannot tolerate a crash or a permanent hardware failure of the machine. Asynchronous checkpointing to stable storage can be used to tolerate such failures. Similar to other replication-based techniques, SoftX cannot tolerate deterministic software bugs or incorrect configurations. A deterministic bug will occur on all replicas and generate the same results. Thus, voting will not detect it. This problem could be solved by using diversity[2](e.g., [7], [28]). SoftX can still detect a wide range of transient faults that corrupt data during transaction execution. These include hardware transient faults and random software bugs which are difficult to find/reproduce (e.g., race condition and other concurrency bugs).

IV. Design and Implementation

SoftX is a software transactional memory system that provides programmers the traditional TM constructs (e.g., atomic block, read, write) for building transactional applications. At the same time, SoftX's architecture ensures resiliency to transient faults. In SoftX, an application thread that starts a transaction, forks a group of threads (on different cores). Each thread in the group executes the same transaction speculatively and independently from others. At the beginning of a transaction, they synchronize their starting points such that all will observe the same initial state.

At commit time, instead of proceeding with the commit operations, the first thread completing its execution contacts a group of dedicated threads, called *committer threads*, for deciding whether the transaction can be committed depending on the outcome of each speculative execution, or must be aborted. The transaction is committed only when the threads' validation succeeds and majority of the speculative executions reach the same stage (i.e., no conflicts and no unmasked faults); otherwise, an abort signal is sent back to the threads.

Committer threads as a whole can be considered as a voter component. Instead of allowing the speculative threads to synchronize with each other for finalizing the commit, relying on the committer threads for the commit operation offloads work from the speculative threads. In addition, having dedicated committer threads reduces cache misses and invalidations (as the overhead of CAS operations is avoided), improving performance, as also shown in [31]. Algorithm 1 and 2 show SoftX's pseudo code[3].

SoftX's design is inspired by NOrec [16]. This choice was made because NOrec uses a single global lock, thus the synchronization between the speculative threads and the committer threads is simple and efficient. Speculative threads proceed similar to NOrec until commit time. Upon reaching the commit stage, each thread alerts committer threads through setting a shared flag for accomplishing the commit procedure

Algorithm 1 Transaction speculative execution

```
1:  procedure txBegin(groupId)
2:      myIndex ← IncAndGet(groupIndex[groupId])
3:      if myIndex = 1 then
4:          SendSignal(No_Commit)
5:      txStartTime ← timestamp
6:      if myIndex = GROUP_SIZE then
7:          SendSignal(Proceed_Commit)
8:          groupIndex[groupId] ← 0
9:  function txRead(addr, groupId)
10:     if WriteSet.contains(addr) then return WriteSet.get(addr)
11:     val ← Memory[addr]
12:     while txStartTime ≠ timestamp do
13:         txStartTime ← timestamp
14:         if ¬Validate() then
15:             AbortGroup(groupId)
16:         val ← Memory[addr]
17:     ReadSet.put(addr, val)
18:     return val
19: procedure txWrite(addr, val, groupId)
20:     WriteSet.put(addr, val)
21: procedure txCommit(groupId)
22:     RequestCommit(myIndex, groupId)
23:     loop
24:         WaitForResponse
25:     if GetResponse() = ABORT then
26:         Restart()
27:     else
28:         FinishGroup()
29: function Validate
30:     if CommittersInWriteBack() then
31:         Wait()
32:     for all entry in ReadSet do
33:         if entry.val ≠ Memory[entry.addr] then      ▷ Value-based
34:             return false
35:     return true
```

(i.e., validation and memory write-back if the transaction is valid). Subsequently, speculative threads wait until the notification of committer threads, and either commit or restart. During the execution, speculative threads log their read and written objects into private memory areas called read-set and write-set, respectively (line 1.17, 1.20).

Speculative threads in charge of executing the same transaction are logically grouped. A group's size can be configured according to the degree of resiliency desired i.e., number of transient faults that can potentially occur in parallel. SoftX decides to commit a transaction if the majority of the speculative executions' outcome coincides. For example, using three speculative threads per group masks a single transient fault without re-execution. If more faults occurred, the speculative execution is restarted to mask these faults.

Figure 1 shows the communication between the speculative threads of a group and the committer threads (or "committers"). In this case, a group has three threads. Each thread executes the same sequence of operations independently from other threads. At the beginning of a transaction, all threads of the group must observe the same initial state (i.e., same starting timestamp). Thus, the first thread, before beginning a new transaction, sends a signal `No_Commit` to committers to pause any commit procedure (line 1.4). This way, the global timestamp cannot change until all threads in the group

[2]Each replica produces the same results via different approaches (e.g., different versions of the same application.)

[3]For the identification of lines in the pseudo-codes, we use the notation Algorithm.Line.

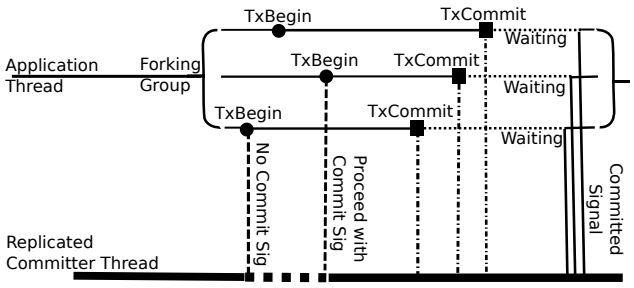


Fig. 1. Processing transaction with SoftX.

successfully start a new transaction (line 1.7). Committers resume the processing of commit requests when the last thread in the group sends the signal, `Proceed_with_Commit` (line 2.20).

In order to detect possible invalidations during transaction processing, any member of a group sends an abort signal to other group members to abort as soon as a previously read object becomes invalid (line 1.15). For this reason, speculative threads validate their read-set after each read operation (line 1.12 - 1.16). This way, all group members restart their execution from the same initial state after an abort.

Algorithm 2 Committer threads

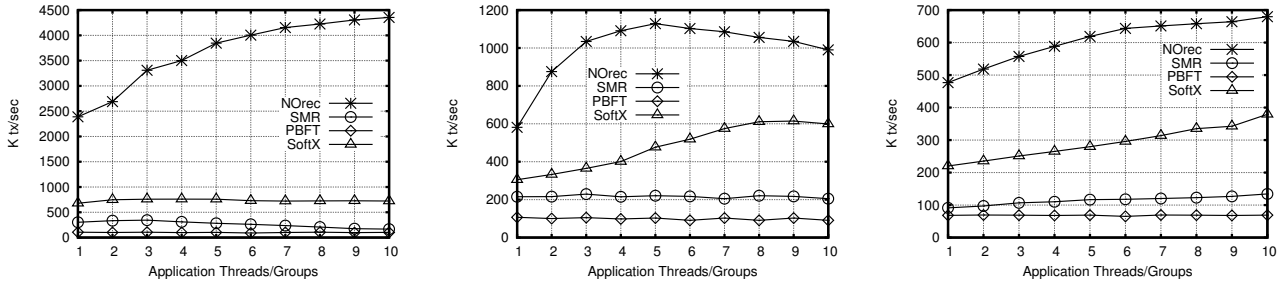
```
1: loop
2:     for group ← 1 to activeGroupsNum do
3:         doneMembers ← GetDoneMembersNum(group)
4:         if doneMembers = GROUP_SIZE then
5:             MySelectedGroup(myIndex, group)
6:                     ▷ Agree on selected group between committer threads
7:             abortNum ← 0
8:             for i ← 1 to GROUP_SIZE do
9:                 if ¬Validate(GetContext(group, i)) then
10:                    abortNum ← abortNum + 1
11:            if abortNum > GROUP_SIZE/2 then
12:                SendResponse(ABORT, group)
13:            else
14:                majority ← Vote(group)
15:                                    ▷ Compare group's write-sets
16:                if ¬ majority then
17:                    SendResponse(ABORT, group)
18:                else if myIndex = CUR_COMMITTER then
19:                    if NO_COMMIT_FLAG then
20:                        WaitForProceedCommitSig(timeout)
21:                    DoWriteBack(majority, group)
22:                    Notify(WRITEBACK_DONE)
23:                else
24:                    WaitForWriteBackSig(timeout)
25:                    if ¬ValidateWriteback() then
26:                        FixCommitters()
27:                    SendResponse(COMMIT, group)
28:        else if doneMembers > 0 then
29:            StartGroupTimeout(group)
```

At commit time, each thread in the group sends a commit request to the committer (line 1.22) and waits for the decision (line 1.24). The committers consist of replicated threads such that they can mask transient faults during the commit procedure. As a result, they act as a "voter." They wait for a commit request from each speculative thread and starts the commit procedure when requests from all group members are received (line 2.4). Then, committers need to agree on which group to commit together (line 2.5). Then, each committer independently starts to validate the read-set generated by the speculative execution of a thread in the group (line 2.9). If the majority of speculative executions is valid (line 2.11), committers compare group members' write-set to ensure that the majority is identical (line 2.14). At this stage, each committer thread has reached a decision on whether to commit or abort its speculative execution, independently. After that, another coordination among committer threads is required to compare their final decisions (line 2.12, 2.17, 2.27). If a majority is reached, one committer thread executes the agreed decision (commit or abort). During write-back operation, an undo log is used to store current values in the memory before overwriting them. Then, in order to tolerate transient faults during this stage, the other committer threads confirm that the values written to memory match the original write-set's values (2.24, 2.25). Otherwise, the undo log is used to restore original memory state (line 2.26).

A transient fault can occur in an application thread, causing it to produce incorrect results or stalling it indefinitely (i.e., it becomes a zombie thread). On the one hand, incorrect results are detected when write-sets are compared and read-sets are validated by the committers. On the other hand, a zombie thread is detected by a timeout while waiting for the commit request from all threads in a group (line 2.29). Speculative threads do not write to shared memory; only the committers can change it. Memory protection mechanism is used to enforce read-only access for speculative threads. For making the committer threads resilient to transient faults, they execute the same steps in an independent manner. Then they compare their outcomes during the coordination phases. Even in this case, a timeout mechanism is used to detect possible transient faults (e.g., line 2.24).

SoftX requires that speculative threads in a group have identical inputs to produce identical behavior. In other words, their actions must be deterministic [8], [10]. To do that, we scope out any form of non-determinism that could result in possible false transient faults. For example, if a transaction calls `random` function, we extract the generation of the random number from the transaction's body, and make this number available for all speculative threads.

Another issue that SoftX has to cope with is the so called *timestamp extension* [16]. This consists of extending the original timestamp of a transaction when it observes that its read-set is still valid but the timestamp has been increased by other commits. This way, subsequent read operations could save the overhead of re-validation if there is no change in the global timestamp (i.e., no transactions committed in between those two read operations). In SoftX, speculative threads act independently, thus some threads could extend the timestamp rather than others. Extending the timestamp synchronously significantly impacts performance. Therefore, we made the decision of extending the timestamp asynchronously. In case some speculative thread arrives at its commit phase with a timestamp different from others and different results, we



(a) Bank (b) List (c) TPC-C

Fig. 2. Transactional throughput of SoftX on x86.

consider it as faulty execution and we abort the entire group. We adopt this "conservative" behavior because the committers cannot distinguish between the case in which the timestamp has been corrupted by a transient fault from the case in which the speculative thread missed extending its timestamp. (We also explored the solution removing the timestamp extension feature. However, the resulting performance was worse than keeping the timestamp extension and aborting the entire group.)

Another important aspect is the methodology for allocating memory slots within a transaction. When a new memory slot is allocated, each speculative thread acquires a different memory location because its area differs from areas allocated by other threads, even though those areas correspond to the same logical object. This results in different values (i.e., memory addresses) in the group's write-sets, and therefore, all members will never have identical write-sets. To overcome this problem and make the validation procedure feasible, we mark each new memory location with a logical reference, and we compare the value of the location instead of the address. Then we verify that the memory address is correct and if it points to the same value. On the contrary, memory deallocation is managed by committers, instead of speculative threads.

V. Experimental Results

SoftX is implemented on two architectures: a shared-bus architecture, represented by common x86 compliant multicore systems, and a message-passing architecture, represented by the Tilera TILE-Gx board [18].

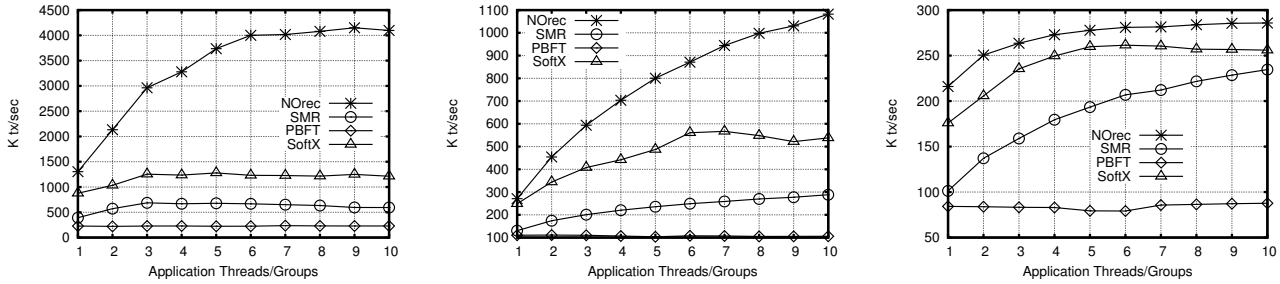
Regarding the former, we conducted our experiments on a 48-core machine, which has four AMD Opteron^{TM} Processors (6164 HE), each with 12 cores running at 1700 MHz, and 16 GB of memory. The machine runs Ubuntu Linux Server 12 LTS 64-bit. Regarding the latter, we used a 36-core board of the Tilera TILE-Gx family. This hardware is commonly used as an accelerator or an intelligent network card. Each core is a full-featured 64-bit processor (1.0 GHz), with two levels of caches, 8 GB DDR3 memory, and has a non-blocking mesh that connects cores to the Tilera 2D-interconnection system.

We implemented our solution using C++ in Rochester STM (RSTM) library [15]. RSTM uses platform-specific assembly instructions. Therefore, we ported its original implementation to support Tilera TILE-Gx family.

In order to assess the performance of SoftX, we used three competitors. The first is the original, non transient faults tolerant, version of NOrec [16]; the second is an SMR approach [12], typically used in transactional replication; and the third is PBFT [6] which represents a byzantine fault-tolerant system.

We implemented an SMR system in a centralized setting inspired by [12]. In this system, each client (i.e., application thread) reserves a core and sends its requests to replicas (or nodes) via shared memory queues or hardware message passing channels. A replica represents a partition of resources in terms of memory and cores. Specifically, each reserves one core for network communications and a variable number of cores to execute transactional requests in parallel. According to the SMR paradigm, clients send their transactional requests to an ordering layer, which is responsible for ordering those requests and delivering them to replicas. With the purpose of keeping consistent the states on all replicas, each must execute transactional requests in the order defined by the ordering layer. We implemented this order by tagging all client requests with the value of a single shared counter using the atomic fetch-and-increment instruction. For respecting the order at the level of the concurrency control, usually, in SMR systems, each replica executes requests sequentially. We overcome this lack, developing a replica concurrency control that supports parallel execution but, at the same time, it enforces the requests' order. Each client also acts as a voter that collect replies from replicas. We added the voter because also this approach has been designed for avoiding transient fault. We only assume that a transient fault cannot happen when the atomic fetch-and-increment instruction is called. PBFT is implemented similarly but it executes transactions sequentially and has a higher overhead in the ordering layer according to PBFT original design.

We tested SoftX on three well-known benchmarks in transaction processing such as Bank, List and TPC-C [17]. Bank mimics operations of a monetary applications. List is a micro benchmark that represents the list data structure. List operations pay the cost of traversing the data structure, increasing the read-set size and thus the execution time. TPC-C is the classical representative of an online transaction processing system based on warehouses and orders that clients perform on shared items. These benchmarks expose different transaction



(a) Bank (b) List (c) TPC-C

Fig. 3. Transactional throughput of SoftX on Tilera.

execution time: in Bank transactions are very short; in List transactions are longer and conflict rate is higher; in TPC-C transactions are the longest because they involve more computation. They differ also in terms of transaction profiles. Bank has one profile for writing bank accounts (i.e., the write transaction) and one for retrieving informations from bank accounts (i.e., the read-only transaction). List has three operations representing the list data structure insert, delete and contains (read-only) operations. TPC-C has five transaction classes: three of them are write transactions while two are read-only transactions. In this evaluation study, we configured Bank for generating 65% of write transactions, List with uniform distribution between the three operations, and TPC-C with its default configuration. All data points plotted are the average of five repeated samples.

As for the STM configuration, in the SMR approach we fixed the number of replicas to three because it matches the size that we used in SoftX as groups' size. Moreover, having only three replicas does not reduce significantly the resource available per replica. For PBFT, we used four replica which is the minimum number of replica required. BFT systems requires $3f + 1$ replicas to tolerate f faults. In the following plots we measure the transactional throughput while we increase the number of groups serving transactions. As a result, when we report the results on 10 groups, it means that the system is running 30 threads. Finally, the number of committer threads is always fixed at three.

A. Shared-bus architecture

Figure 2 shows the results on x86 shared-bus architecture. In all benchmarks, the shared-bus represents a bottleneck for replication-based systems. SoftX minimal synchronization improved its performance compared to replication-based systems. It also reduced the overhead of SoftX compared to non faults tolerant system.

Figure 2(a) shows the results with Bank benchmark. Performance of NOrec is $4\times$ better than SoftX. This large overhead is due to Bank's very small transactions, which magnify the synchronization overhead of SoftX. In addition, NOrec does not have any mechanism for being resilient to transient faults. However, when compare to the SMR approach, SoftX does not suffer from the overhead of global ordering and in-order processing, thus outperforming it by $1.96\times$ on average. Compared to PBFT, SoftX outperforms PBFT by $6.3\times$ on average. PBFT has higher overhead to reach consensus and also executes requests sequentially.

Figure 2(b) shows the results with the List application. As the length of list's transactions is longer and the contention is higher, the additional overhead due to multiple synchronization points, needed for tolerating possible transient faults, is less evident than in Bank. Thus, SoftX overhead is reduced to $1.15\times$ on average compared to NOrec. When the contention in the system increases due to multiple threads working, performance becomes more comparable (65% overhead) because SoftX exploits the presence of committer threads, which reduces also hardware contention and increase system's scalability. As a general trend, SoftX performs better than SMR. The reason is mostly related to the in-order processing and also the final voting that clients perform at the end of each transaction. SoftX optimizes this process: speculative threads process in parallel to committer threads. This way the transaction execution in the system is overlapped with the commit phase, thus reducing the stalls that happen on the SMR approach. SoftX outperforms SMR by $1.23\times$ on average. SoftX outperforms PBFT by $3.89\times$ on average.

Figure 2(c) shows the results with TPC-C benchmark. Here the benchmark is complex and transactions are the longest compared to List and Bank. In terms of contention, TPC-C has lower contention than List as we use 100 warehouses whereas in List, all transactions traverse the same nodes of the list, which increase the conflicts. NOrec outperforms SoftX by $1.1\times$ on average. SoftX gains up to $1.54\times$ compared to SMR and $3.27\times$ compared to PBFT.

Summarizing, SoftX overhead is acceptable for application that has high contention (e.g., List) and/or long transactions (e.g., TPC-C).

B. Message-passing architecture

Figure 3 shows the results on the Tilera message-passing architecture. With the more communication bandwidth, all fault-tolerant approaches improve their performance.

Figure 3(a) shows the results with Bank benchmark. Performance of NOrec is now $1.8\times$ better than SoftX compared to $4\times$ in shared-bus results. The message-passing network increased the communication bandwidth between different cores and now synchronization overhead is distributed over

multiple links. For the same reason, replication-based approaches overhead is decreased as it is designed for exploiting networking capabilities. SoftX outperforms SMR by $0.94\times$ on average. And SoftX outperforms PBFT by $4.18\times$ on average.

Figure 3(b) shows the results with the List benchmark. The overhead of SoftX is farther decreased given List's high contention nature and message-passing's higher bandwidth. SoftX overhead is reduced to 61% on average compared to NOrec. SoftX outperforms SMR by $1.02\times$ on average. SoftX outperforms PBFT by $3.37\times$ on average.

Figure 3(c) shows the results with TPC-C benchmark. Given the TPC-C's long transactions, SoftX overhead is minimal to 12% only on average compared to NOrec. SoftX is 33% better compared to SMR and $1.88\times$ better compared to PBFT.

In summary, having a message-passing architecture, synchronization and communication overhead is significantly reduced. Now, the network bandwidth does not represent a bottleneck in most of the benchmarks. The benefits of SoftX's and SMR's concurrent execution are evident compared to PBFT, which does not scale with increased number of threads. SoftX still outperforms replication-based approaches.

SoftX performs better than replication-based approaches since it requires less data transfer between system components. State-machine replication is designed for distributed systems which have a network. In x86 centralized system, using memory queues as a network saturates the shared bus and reduces the performance, significantly. Moving the implementation to a message-passing architectures reduces replication-based systems' overhead however, even in this case, SoftX outperforms the other competitors.

VI. Conclusions

SoftX confirms that it is possible providing a concurrency control protocol that makes transactional applications resilient to transient faults, without giving up high performance. We leveraged the huge amount of computational resources available in recent multicore architectures, as well as a protocol that limits the overhead by combining speculative execution and dedicated committer threads. Our results show lower overhead compared to replication-based approaches even with an optimized state-machine replication protocol. SoftX is also suitable for current shared-bus architectures, as well as the emerging message passing architectures.

Acknowledgement

This work is supported in part by US National Science Foundation under grant CNS-1217385.

References

[1] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, vol. 25, no. 6, 2005.

[2] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *Device and Materials Reliability, IEEE Transactions on*, vol. 5, no. 3, pp. 305–316, 2005.

[3] F. B. Schneider, *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., 1993.

[4] S. Peluso, P. Romano, and F. Quaglia, "Score: A scalable one-copy serializable partial replication protocol," in *Middleware*, 2012.

[5] M. Wiesmann, A. Schiper, F. Pedone, B. Kemme, and G. Alonso, "Database replication techniques: A three parameter classification," in *SRDS*, 2000, pp. 206–215.

[6] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM TOCS*, vol. 20, no. 4, 2002.

[7] B.-G. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine-fault tolerance." in *USENIX Annual Technical Conference*, 2008, pp. 287–292.

[8] R. Palmieri, F. Quaglia, and P. Romano, "Osare: Opportunistic speculation in actively replicated transactional systems," in *SRDS*, 2011, pp. 59–64.

[9] ——, "Aggro: Boosting STM replication via aggressively optimistic transaction processing," in *NCA*, 2010, pp. 20–27.

[10] S. Hirve, R. Palmieri, and B. Ravindran, "Hipertm: High performance, fault-tolerant transactional memory," in *ICDCN*, 2014, pp. 181–196.

[11] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, 2003.

[12] M. Mohamedin, R. Palmieri, and B. Ravindran, "Managing soft-errors in transactional systems," in *DPDNS*, 2014.

[13] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *ICDCS*, 2012, pp. 455–465.

[14] N. Shavit and D. Touitou, "Software transactional memory," in *PODC '95*.

[15] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott, "Lowering the overhead of nonblocking software transactional memory," in *TRANSACT*, 2006.

[16] L. Dalessandro, M. F. Spear, and M. L. Scott, "NOrec: Streamlining STM by Abolishing Ownership Records," in *PPoPP '10*.

[17] T. Council, "TPC-C benchmark," 2010.

[18] Tilera Corporation, *TILE-Gx Processor Family*, http://www.tilera.com.

[19] G. Lu, Z. Zheng, and A. A. Chien, "When is multi-version checkpointing needed?" in *FTXS*, 2013.

[20] L. L. Pullum, *Software fault tolerance techniques and implementation*. Artech House, 2001.

[21] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini, "Practical hardening of crash-tolerant systems," in *USENIX Annual Technical Conference*, 2012.

[22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," ser. SOSP, 2007.

[23] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden, "Tolerating byzantine faults in transaction processing systems using commit barrier scheduling," ser. SOSP, 2007.

[24] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: Execute-verify replication for multi-core servers," ser. OSDI, 2012.

[25] S. Kim, M. Z. Lee, A. M. Dunn, O. S. Hofmann, X. Wang, E. Witchel, and D. E. Porter, "Improving server applications with system transactions," ser. EuroSys, 2012.

[26] D. Bernick, B. Bruckert, P. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen, "NonStop advanced architecture," ser. DSN, 2005.

[27] J. Pool, I. S. K. Wong, and D. Lie, "Relaxed determinism: Making redundant execution on multiprocessors practical." in *HotOS*, 2007.

[28] Y. Zhou, D. Marinov, W. Sanders, C. Zilles, M. d'Amorim, S. Lauterburg, R. M. Lefever, and J. Tucek, "Delta execution for software reliability," ser. HotDep, 2007.

[29] A. Turcu, B. Ravindran, and R. Palmieri, "Hyflow2: a high performance distributed transactional memory framework in scala," in *PPPJ*, 2013, pp. 79–88.

[30] N. L. Diegues and P. Romano, "Time-warp: lightweight abort minimization in transactional memory," in *PPOPP*, 2014, pp. 167–178.

[31] J.-P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications," ser. USENIX ATC'12, 2012.

[32] A. Hassan, R. Palmieri, and B. Ravindran, "Remote invalidation: Optimizing the critical path of memory transactions," in *IPDPS*, 2014.