On Preserving Data Integrity of Transactional Applications on Multicore Architectures

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 increasingly becoming prone to transient faults. In this paper we briefly present Shield, a middleware to provide transactional applications with resiliency to those faults that can happen anytime during the execution of a processor but do not cause any hardware interruption. Shield is inspired by the state machine replication approach, where computational resources are partitioned, the shared state is fully replicated, and requests are executed by all partitions in the same order. Shield embeds a set of algorithmic and system innovations to limit the overhead with respect to non-fault-tolerant solutions. They include a fast total order layer that lets application threads and computational nodes co-operate in order to fast deliver.**

I. Introduction

The proliferation of multicore architectures defines the current technological trend and, in such systems, the problem of tolerating/detecting data corruption is complex due to the nature of the underlying hardware [1]. As an example, *soft-errors* [2] belong to the category of hardware-related errors that are difficult to detect or predict. Specifically, they are transient faults that may happen anytime during the application execution. They are caused by physical phenomena [1], e.g., cosmic particle strikes or electric noise, which cannot be directly managed by application designers or administrators. As a result, when a soft-error occurs, the hardware is not affected by interruption, but applications may behave incorrectly.

In this paper we focus on those soft-errors that occur inside the processor (which we name hereafter CPU-TFAULTs). That is because CPU-TFAULTs are random, hard to detect, and can corrupt data – e.g., a CPU-TFAULT can cause a single bit to flip in a CPU register due to the residual charge of a transistor, which inverts its state. Most of the time, such an event is likely to be unnoticed by applications because they do not use that value (e.g., an unused register). However, sometimes, the register can contain a memory pointer or a meaningful value. In those cases, the application behavior can be unexpected. An easy solution for recovering from transient faults is a simple application-restart, but for those applications with stringent reliability requirements, which form the focus of this paper, it cannot be acceptable.

Motivated by these observations, we propose *Shield*, a software middleware for tolerating CPU-TFAULTs. Shield's basic scheme is inspired by the state machine replication approach (SMR) [3], where computational resources are partitioned, data are replicated across partitions, and application requests are executed on all partitions following the same (previously agreed) order. The goal of Shield is to prevent any data stored in a processor register from being propagated to the system's main memory where the shared state is kept, without being verified as free of corruption. Shield is designed for applications where many threads act on the same shared state and where the data integrity is fundamental

As in the SMR approach, Shield processes transactions in the same order on all replicated states. This redundant execution isolates any possible propagation of incorrect transition to the memory without being previously certified. The latter operation is performed by a *voter*, a software module that collects the outcome of transactions and delivers the common response (i.e., the majority of replies) to the application. The voting outcome is also used for identifying corrupted computations, and, if so, other correct states are exploited for overwriting the broken parts of the memory. As the other software components of Shield, also the voter is not assumed to be reliable.

II. Is Byzantine Fault Tolerance the Solution?

CPU-TFAULTs are transient faults, and that class of faults belongs by itself to the category of *Byzantine Faults* (BF) [4]. A BF is an arbitrary fault that can generate incorrect response or corrupt the system state. BFs include commission and omission faults. Solutions targeting BFs, named also *Byzantine Fault Tolerant* (BFT) protocols (e.g., [5], [6]), are usually designed for minimizing the assumptions on the correctness and trustiness of components composing the execution environment, as well as for being resilient to malicious behaviors. Given that, the answer to the above question is clearly: *yes*, BFT is a solution solving also CPU-TFAULTs but it is very "pricy".

In fact, deploying a BFT solution would have an impact on the system's performance much higher than what is actually needed for solving the problem of CPU-TFAULT. In addition, BTF solutions often require a physical multi-node distributed system to isolate nodes from each other and therefore avoid the propagation of faults. However, replicating centralized systems for tolerating faults results in significantly degraded performance (e.g., 10–100$\times$). That is primarily due to the costs for remote synchronization and communication that are incurred to ensure node consistency. Also, a distributed architecture comprising of multicore nodes may not often be cost-effective. BFT systems handle malicious client behavior and unreliable networks that can reorder, drop, or corrupt messages. In addition, most BFT solutions require $3f + 1$ nodes to reach agreement and $2f + 1$ nodes for the transaction execution in order to tolerate up to f faulty nodes [7].

Shield is meant to be a software layer that can be plugged into a classical centralized transactional system without deploying a distributed infrastructure or substantially impacting the performance of the original system. Its design assumes trusted clients and a reliable network infrastructure. We believe that adopting a BFT solution for solving the problem of CPU-TFAULTs would be "inaccurate" because of the excessive negative performance penalty that the application has to pay.

III. Assumptions

We consider a system based on the message-passing abstraction where a set of *nodes*, installed on the same physical hardware, communicate with each other through a reliable FIFO channel (e.g., the bus). We consider each node as a group of computing cores on a multicore board. We assume the presence of a service providing a single monotonically increasing clock, which we name *clock-service*. In practice, some message-passing boards are already equipped with such a service through special hardware extensions; and on x86 architectures, there are techniques (as [8]) that exploit the non-synchronized hardware register that stores the CPU-cycle counter to provide the clock-service as defined above.

Application's shared state is replicated such that each node accesses its own copy. This way, storing a value from a CPU register to a memory location does not interfere with the work of other nodes, which prevents the propagation of a possible fault to other nodes. To tolerate f CPU-TFAULTs, Shield requires $2f + 1$ nodes so that a majority can be formed and the voting procedure can take place.

IV. Shield Overview

At a glance, our ordering protocol involves application threads (because they are physically located together with all other nodes) and it does not rely on a single component to order (e.g., the sequencer). When a transaction is requested to execute by an application thread, it is sent to all nodes and other application threads, together with the current timestamp taken from the clock-service. This timestamp represents a tentative order for executing the transaction (which we name *tn-order*) [9]. To determine its total order, a node must ensure that it receives a request from each application thread with a timestamp that is higher than the one just received. (For this reason we said above that our ordering layer involves application threads.) When a node receives a message from all application threads, it can now safely determine the next transaction to deliver and its total order.

An ordering-based concurrency control (ObCC) protocol, running locally at each node, is responsible for processing and committing transactions on the node's private memory. At this stage, the application thread is still not informed about the transaction outcome because an additional *voting* phase (see below) is required. A subset of cores in each node is dedicated for the execution of ObCC. ObCC leverages the tn-order for anticipating the transaction processing. We name this execution as "speculative" because the tn-order is tentative and can be contradicted by the total order. Also, in order to maximize the overlapping of the transaction processing with the establishment of the total order, transactions are processed in parallel (using the tn-order for solving conflicts).

A *voter* is in charge of collecting transaction outcomes from all nodes and returning the majority of them to the application. Even though this approach potentially increases the end-to-end transaction latency because the voter has to wait for a majority of outcomes, nodes are part of the same architecture thus their progress is likely not skewed. Using a lazy concurrency control, where operations are buffered until reaching the commit phase (as in our ObCC), simplifies the comparison procedure of the voter because each transaction records all its accessed objects into private memory spaces.

When the voter restarts a faulty node, Shield enters into *recovery mode* and starts overwriting the state from a correct node. The copy is incremental: the non-faulty node keeps track of all objects modified during the copying process so that it can still serve new transactions. This incremental state is then pushed to the faulty node for finalizing the copy.

V. Voter

The voter component consists of $2f+1$ voter threads so that even if a CPU-TFAULT occurs during the verification process, no wrong decision can be made. Each thread independently compares the outcomes of the next-to-commit transaction, according to the total order, by matching the gathered read-sets and write-sets from all nodes. When an error is detected (i.e., there is no matching), a voter thread sends to the faulty node a recovering signal. A faulty node starts the recovery process only upon receiving $f + 1$ recovery requests, which guarantees that the error actually happened on the node and not on the voter thread. Following the same policy, each voter thread compares the decisions of all other voter threads to confirm its decision matches the majority. Finally, one non-faulty voter thread sends the result to the application thread that originated the request.

Acknowledgment

This work is supported in part by US National Science Foundation under grant CNS 1217385 and by AFOSR grant FA9550-14-1-0163.

References

[1] 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.

[2] S. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *Micro, IEEE*, 2005.

[3] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

[4] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, pp. 382–401, 1982.

[5] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, 1999.

[6] B.-G. Chun, P. Maniatis, and S. Shenker, "Diverse replication for single-machine byzantine-fault tolerance," ser. ATC, 2008.

[7] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," ser. SOSP, 2003.

[8] W. Ruan, Y. Liu, and M. Spear, "Boosting timestamp-based transactional memory by exploiting hardware cycle counters," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 40:1–40:21, Dec. 2013.

[9] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE TKDE*, vol. 15, no. 4, 2003.