RETROSPECTIVE: Transient-Fault Recovery for Chip Multiprocessors

Mohamed Gomaa
MComp IT Solutions
mohamed.a.gomaa.m@gmail.com

Chad Scarbrough
Cisco
cscarbro@cisco.com

T. N. Vijaykumar and Irith Pomeranz
Elmore Family School of Electrical and Computer Engineering
Purdue University
{vijay,pomeranz}@purdue.edu

Our paper, titled *Transient-Fault Recovery for Chip Multiprocessors* [2], tackled the problem of increased susceptibility of scaled technology nodes to transient faults. (Multicores is the current term for chip multiprocessors.) The paper proposed *Chip-Level Redundantly Threaded Multiprocessor with Recovery (CRTR)* which runs two copies of each thread, one ahead of the other, the leading and trailing copies. In prior work on handling transient faults in single, hardware-multithreaded cores, the two copies run in different hardware contexts on the same simultaneously-multithreaded (SMT) core (e.g., Active-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) [9], Simultaneous and Redundant Threading (SRT) [8], and Simultaneous and Redundant Threading with Recovery (SRTR) [10]). While the leading thread is the original thread which includes speculative instructions, the trailing thread uses the leading thread's branch-prediction outcomes to execute only the correctly-predicted instructions which are around 45-60% of all speculative instructions. Further, the trailing thread overlaps its execution with the leading thread's cache misses incurring much lower slowdowns than that of naively redundant execution.

In a multicore, the two copies run on different cores, as proposed in Chip-Level Redundantly Threaded Multiprocessors (CRT) [5]. Unlike detection schemes which compare only the store addresses and values of the two copies [5], [8], recovery schemes compare the copies' register values before instruction commit because committing an erroneous register value – a case of silent data corruption – would make recovery impossible [2], [10]. However, the register values impose high inter-core bandwidth pressure in multicores, even if the copies run on adjacent cores.

To address this issue, CRTR reduces the bandwidth demand apart from pipelining the communication path for higher bandwidth supply. Because faults propagate through (true) dependencies, our previous work [10] proposed *dependence based checking elision (DBCE)* so that only the output of the last instruction in a dependence chain is checked. However, instructions that mask operand bits may mask faults so that a faulty register value may be committed if the masking instruction is not at the end of its dependence chain. On the other hand, ending the chain at such masking instructions may limit the chain length and bandwidth savings. Instead, we made the key observation that if an instruction's source operand dies at the instruction then any masked fault in the operand does not corrupt later computation. Accordingly, we proposed *death-and dependence-based checking elision (DDBCE)*, which chains a masking instruction only if the source operand of the instruction dies at the instruction.

CRTR's results show that CRT incurs around 15% single-thread slowdown compared to a non-fault-tolerant multicore, and CRTR does not worsen this degradation even for 30-cycle adjacent-core communication latency. Further, CRT requires 5.2 bytes per cycle adjacent-core communication bandwidth, and CRTR with DDBCE increases the demand only to average 7.1 bytes per cycle.

CRTR has had significant impact on hardware fault-tolerance research, accruing more than 480 citations till date. CRTR has spurred considerable follow-on work, including our own [3], [6].

Continued technology scaling means ever-increasing susceptibility not only to transient faults but also to aging-related hard faults. However, transient faults affecting compute logic in real-world multicores have not been reported widely, though memories – DRAM and multi-megabyte on-chip SRAM caches – continue to need to be protected by ECC against many types of faults, including transient ones. On the other hand, data center operators have reported failures traced to aging-related hard faults as the root cause [1], [4].

Though proposed for soft errors, CRTR's ideas can be applied to hard faults as well. For instance, in distributed, multithreaded or multi-process applications running on shared- or distributed-memory systems, merely detecting faults (for instance, via CRT) is not enough. Even though stores are checked in CRT to prevent memory from being corrupted by a fault, corrupted register data would render the corresponding thread or process irrecoverable. Further, the faulty thread's memory state though not corrupted may be inconsistent globally across the other threads or processes because the faulty thread may have updated memory with correct but incomplete computation that leaves memory globally-inconsistent.

Of course, transaction-based software (e.g., SQL databases) can recover from such problems by reverting to a previous globally-consistent state. However, such software is non-trivial and expensive to develop and maintain, and is relied upon only for the most stringent contexts (e.g., financial and medical accounts). In other contexts (e.g., widely-used noSQL databases which do not provide transactional behavior), faults can create serious system-wide inconsistencies. Such errors can be prevented by CRTR without software complexity. For hard faults, a mismatch in the two copies in CRTR indicates the presence of a fault but does not identify the faulty core. To that end, both cores are suspended upon a fault-detection exception and one of the copies is migrated to a third core via standard thread migration. Then, both copies resume execution for as many instructions as the instruction window of the core (e.g., 500-1000). Because the error was caught in the original run before being committed, the error was triggered within the instruction window. Thus, the second execution with the migration will lead to a mismatch within the window, if the non-migrated core was faulty. Otherwise, the core from which the migration occurred is faulty. The OS has to disable the faulty core. In the unlikely event of multiple core failures, the OS can handle the ensuing fault-detection exceptions one at a time. The application threads can continue to execute on the remaining non-faulty cores.

For aging-related hard faults, enabling CRTR's redundancy all the time may be undesirable for performance reasons, especially when the system has not aged. As such, as a trade-off between performance and reliability depending on the age of the system, CRTR can be enabled for some contiguous stretch of time periodically at a rate determined by the OS eventually resorting to continuous checking for some of the cores depending on their ages.

Finally, transient and hard faults are a serious issue in the emerging domain of self-driving cars where fault recovery may be crucial for continued operation. The above migration-based CRTR scheme can ensure uninterrupted operation. Software-based self-test approaches that directly target faults can further improve reliability when run under CRTR [7].

References

[1] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. Silent data corruptions at scale. *CoRR*, abs/2102.11245, 2021.

[2] Mohamed Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, page 98–109, New York, NY, USA, 2003. Association for Computing Machinery.

[3] Mohamed A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, page 172–183, USA, 2005. IEEE Computer Society.

[4] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, page 99–110, USA, 2002. IEEE Computer Society.

[6] Nitin, Irith Pomeranz, and T. N. Vijaykumar. Faulthound: Value-locality-based soft-fault tolerance. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 668–681, New York, NY, USA, 2015. Association for Computing Machinery.

[7] P. Parvathala, K. Maneparambil, and W. Lindsay. Frits - a microprocessor functional BIST method. In *Proceedings. International Test Conference*, pages 590–598, 2002.

[8] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, page 25–36, New York, NY, USA, 2000. Association for Computing Machinery.

[9] Eric Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, FTCS '99, page 84, USA, 1999. IEEE Computer Society.

[10] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, page 87–98, USA, 2002. IEEE Computer Society.