Asymmetric Resilience: Exploiting Task-level Idempotency for Transient Error Recovery in Accelerator-based Systems
&
Architectural Core Salvaging in a Multi-Core Processor for Hard-Error Tolerance

Yash Shah (ys562)

November 2024

1 Asymmetric Resilience (AR)

AR seeks to provide a **general** and **efficient** transient error recovery solution suiting systems incorporating accelerators, which may be less reliable. Prior work [1, 2] has shown GPU's MTBF is almost $\times 8$ lower than CPU's.

1.1 Related work

Simple checkpointing is inefficient, as storing the accelerator's state before every use and restoring upon error incurs substantial overhead ($\times 10$ slowdown). Epoch-based checkpointing [3, 4] amortizes the checkpointing cost over multiple task runs. However, since an epoch often includes CPU+accelerator use, the checkpointing state space now includes CPU's state too, incurring a large 10–100% overhead.

Idempotency-based checkpointing [5] relies on lack of write-after-read (WAR) dependencies in a code region. Failing idempotent regions can be re-executed for recovery, avoiding costly checkpoints. However, this approach lacks generality—prior work assumes a strong failure model (error detection occurs before corrupting memory) and the technique often requires CPU-exclusive features.

1.2 Implementation

AR is composed of resilient domains and task-level idempotence. Domains ensure safety/isolation of CPU memory. Discrete memory systems provide this isolation by default. For integrated (shared) memory systems, Border Control [6] is used to control every GPU memory access missing the GPU cache.

A task's memory regions are classified into read-only input, write-only output, and input/output. A task (e.g. GPU kernel) is considered idempotent if it consists solely of input, full output, and WAR output memory. Thus, idempotent tasks can use cheap re-execution for recovery, instead of costly checkpointing. Partial writes are disallowed as erroneous execution could corrupt a bit in the output memory which is not overwritten during subsequent re-execution. Approximate, safe static analysis detects idempotency of tasks, classification of memory regions and distinguishing full/partial output.

The key idea is that checkpoints are required only for non-idempotent tasks, significantly reducing normal execution overhead. Upon error, recovery for idempotent tasks is achieved through simple re-execution from the last checkpoint, according to a relaunch set.

This assumes that the expected value of the reexcution time over the error probability distribution is less than the fixed checkpointing overhead. For long idempotent chains, this may not be the case, and thus heuristics are employed to insert checkpoints to maintain optimal chain length. The kernel cohesion optimization is further added by noticing that only input/output and partial memory require checkpointing for non-idempotent tasks.

2 Architectural Core Salvaging (ACS)

Manufacturing-time hard faults are uniformly distributed across the die. Repetitive structures such as caches are easier to protect than complex cores lacking natural redundancy, leading to core disabling or core sparing, costing performance and die area.

ACS observes that a die as a whole can still be ISA-compliant even when some cores cannot execute certain operations, relying on cross-core redundancy to execute troublesome operations through thread migration to a functional core.

2.1 Related microarchitectural salvaging work

Rescue [7] alters the core design to better exploit microarchitectural redundancy, but makes generous assumptions like equivalency and thus redundancy of all decoders. Core cannibalization [8] allows a core to borrow another core's resources at a pipeline-stage level, but this leads to close coupling and requires additional interconnect. Both suffer from complexity and performance issues. Microarchitectural techniques requires a different salvaging mechanism for every structure, costing complexity, die area, and money.

2.2 Implementation

A faulty core operates as normal until a troublesome instruction is detected during instruction decode. Detection may be facilitated through a look-up table fused at manufacture-time, or programmed. The thread must then be migrated

to (/swapped with another thread) from a functional core according to a migration policy e.g. round-robin. Thread migration is facilitated through deep-sleep power-states.

As optimization, migration counts over a clock-cycle window can be maintained and the defective core disabled upon exceeding a threshold, preventing thrashing when all threads utilize a defective structure often. Also, the faulty core can be disabled when fewer than maximum number of threads are available to execute.

A hybrid solution with microarchitectural redundancy is also explored to cover structures critical for pipeline functionality.

3 Comparison

3.1 Complexity of implementation and additional hardware requirements

AR is reliant on both compiler support to provide static analysis for detecting task-level idempotency as well as runtime support error recovery. Static analysis could potentially pose a barrier to AR adoption with large libraries, but the small runtime overhead ($< 1\%$) poses few issues. Additionally, it requires a modified memory subsystem, which may require non-trivial modification in other accelerator architectures driving up implementation-complexity significantly, but the paper only concerns itself with GPUs.

Meanwhile, ACS is mostly transparent to the OS and user, only requiring a programmable APIC ID. The clever use of deep sleep states to transfer threads ensures simplicity as limited hardware modification is required. However importantly, achieving high coverage requires a hybrid approach incorporating microarchitectural redundancy for covering pipeline-critical structures. The small, redundant structures added are bespoke to the structures they are protecting, significantly increasing design complexity.

3.2 Focus on hetereogeneous vs. homogeneous systems

AR tries to provide fault tolerance in hetereogeneous systems where accelerators are assumed to have lower reliability. It has a bias for massively-parallel devices with large state, which makes checkpointing expensive. Checkpointing architectural state on CPUs is cheap, so AR provides little utility.

Contrarily, ACS cannot be applied to manycore architectures. Firstly, it encounters trouble scaling (> 16 cores in the ACS paper) as the loss in throughput by the failure of a core is negligible compared to the thread-transfer overhead and the overhead is too high for low core-counts.

Secondly, in massively-parallel devices like GPUs, all cores are running the same instructions, making ACS resort to core-disabling as all threads frequently encounter troublesome operations. ACS is heavily dependent on minimizing the critical set of instructions to get some use out of a faulty core. However, since

accelerators are domain-specific, they often only implement what is critical per core anyway, making ACS fundamentally incompatible.

3.3 Resilience to soft vs. hard errors

AR is designed to handle transient faults like soft errors and voltage noise. Its lightweight re-execution-based recovery mechanism fundamentally assumes that eventually the hardware will produce the correct result.

ACS focuses on hard errors, especially those residing in combinational logic. It assumes that faults are detected at manufacture-time, so that the correct look-up table can be fused into the core's front-end and appropriate redundant hardware added for the hybrid microarchitectural approach. AR and ACS are necessarily confined to their error domains by design, but their orthogonal nature permits a hybrid approach between the two, interesting for future work.

Both approaches also assume that errors are rare to amortize the cost of re-execution and thread migration respectively. When errors are common, both techniques resort to either thrashing, or core-disabling in the case of ACS.

3.4 Sensitivity to different system configurations

In theory, AR should behave well even in systems with a large number of GPUs and a single CPU, assuming sufficiently low runtime overhead. A slight concern is that if the CPU is in-charge of re-execution, this may lead to latency issues in restarting kernels if GPU failures are frequent. However, since the study does no experiments on other accelerator types, behaviour on non-GPU systems is unknown.

ACS is quite sensitive to core counts. For example, if just a single faulty core loses 25% of its performance, ACS is only suitable for systems with 6–16 cores as the thread migration overhead becomes harder to amortize. This limits ACS's scope to a middling core-count systems. Thread migration also means that ACS is limited to single-socket systems, as cross-socket communication latency is presumably far too high.

3.5 Limitations of evaluation methods

AR's evaluation uses real GPU experiments and simulation for integrated CPU-GPU systems. However, only an analytical model is provided for non-GPU accelerators. This is a severe limitation as its reliance on static analysis may not generalize to other accelerator types (perhaps the approximate analysis deems all memory regions as I/O) and task-level idempotency may not be possible with most accelerator-types. Furthermore, the cost of checkpointing may simply be cheaper than re-execution on certain task+accelerator combinations.

ACS's analysis is limited as it limits itself to single-core faults. In modern high core-count systems, it may be the case that several cores are faulty. Perhaps the thread migration overhead grows quickly as the faulty core-count increases.

4 Future directions for AR

While AR relies on static analysis to detect task-level idempotency and re-execute idempotent chains, it does not fuse kernels as that would require re-compilation. More advanced static analysis could allow multiple idempotent kernels to become one, greatly reducing kernel launch overhead and memory traffic between CPU and accelerator.

There is also potential to combine AR's resilience domains with ACS. A faulty core could be considered to be within a weak-resilience domain and functional cores within the strong one. Tasks that are deemed by a scheduler to be low-criticality or non-latency-critical can be moved onto a faulty core. The resilience domains ensure that the faulty core cannot corrupt the state of functional cores. If an operation cannot be executed on a faulty core, thread migration can take place again. With error-checking of the output, this hybrid approach could make good use of faulty cores.

5 Future directions for ACS

An important issue with ACS is that for a faulty core to be useful, it must still implement some critical operations. There are likely many defective cores which do not meet this requirement and so future work can look at how such cores can play a supporting role for functional ones [9]. One way this could be done is for faulty cores to provide branches and hints wherever possible and then performing a thread migration when no more progress can be made. This has even more potential for modern heterogeneous CPUs employing architectures similar to big.LITTLE, where e.g. a faulty faster core could help a functional, slower one with ALU operations.

In the case of multiple faulty cores, they could potentially be grouped together to form a complementary core where one can perform operations the other cannot. Thread migration could then be restricted to only within this set to avoid impairing the performance of functional cores.

ACS does not scale well with higher core counts. However, modern large core-count CPUs consist of multiple core-complexes which are joined together with high-bandwidth interconnect. Applying ACS at the core-complex level could help salvage more cores, without it leaving its optimal performance zone.

References

[1] Devesh Tiwari et al. "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation". In: *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 2015, pp. 331–342. DOI: 10.1109/HPCA.2015.7056044.

[2] Catello Di Martino et al. “Lessons Learned from the Analysis of System Failures at Petascale: The Case of Blue Waters”. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2014, pp. 610–621. DOI: 10.1109/DSN.2014.62.

[3] Antonio J. Peña, Wesley Bland, and Pavan Balaji. “VOCL-FT: introducing techniques for efficient soft error coprocessor recovery”. In: *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807640.

[4] Hiroyuki Takizawa et al. “CheCUDA: A Checkpoint/Restart Tool for CUDA Applications”. In: *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*. 2009, pp. 408–413. DOI: 10.1109/PDCAT.2009.78.

[5] Shuguang Feng et al. “Encore: Low-cost, fine-grained transient fault recovery”. In: *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2011, pp. 398–409.

[6] Lena E. Olson et al. “Border control: Sandboxing accelerators”. In: *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015, pp. 470–481. DOI: 10.1145/2830772.2830819.

[7] Ethan Schuchman and TN Vijaykumar. “Rescue: A microarchitecture for testability and defect tolerance”. In: *32nd International Symposium on Computer Architecture (ISCA’05)*. IEEE. 2005, pp. 160–171.

[8] Bogdan F. Romanescu and Daniel J. Sorin. “Core Cannibalization Architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults”. In: *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 2008, pp. 43–51.

[9] Amin Ansari et al. “Necromancer: enhancing system throughput by animating dead cores”. In: *SIGARCH Comput. Archit. News* 38.3 (June 2010), pp. 473–484. ISSN: 0163-5964. DOI: 10.1145/1816038.1816024. URL: https://doi.org/10.1145/1816038.1816024.