It's Time for New Programming Models for Unreliable Hardware

Hajime Fujita*†, Robert Schreiber‡, and Andrew A. Chien*†
*University of Chicago †Argonne National Laboratory ‡HP Labs
hfujita@uchicago.edu, rob.schreiber@hp.com, achien@cs.uchicago.edu

I. Introduction

Virtually all computer programming models presume reliable hardware for both their computation and data structures. Even today, hardware experiences many errors, most of which are masked by integrated error correction mechanisms [1], [2], [3]. But once the errors exceed the capability of hardware correction (e.g. an "uncorrectable" error in DRAM), the system simply gives up execution and resets the system. This works well today, and so long as the resulting number of computation failures is small.

In future hardware, as the benefits of Moore's Law wane, the industry is scaling voltages aggressively to reduce power, producing rising error rates both in logic and memory [4], [5]. These increases threaten our ability to reliable hardware at the desired performance and power, and **give rise to the prospect that future software will have to be designed for unreliable hardware. How to do this is challenging; what to do if variables in registers or memory can change silently? How can one build a working program?**

Fault-tolerance in software has been pursued for decades for mission-critical applications through replication, quorum voting techniques, and checkpoint-restart, but these systems are mainly designed to tolerate fail-stop of processes, not to correct errors. In the future, we aspire to a much finer-grained error detection and recovery, tolerating the computation and data errors at the level of single words or a few instructions. **We believe that meeting these challenges requires programming model support in the programming model for detecting and recovering from errors**.

We argue that **future programming models provide support for dealing with hardware errors** conveniently and efficiently. In contrast to techniques proposed in conjunction with specific hardware [6], we believe that programming models for unreliable hardware **must be portable**. If applications make significant investment in programming for errors, then this investment must be preserved across different hardware platforms. This portable programming model support must allow programs to:

- capture hardware detected and signalled errors,
- capture operating system detected errors,
- express computation error checks, and
- respond to errors when those checks fail.

We are not the first to raise this issue, but our advocacy of this idea in a mainstream context for software is unique. Efforts such as Recovery Blocks [7] were designed for highly-reliable systems, and Relax include reliability-relaxing programming extensions to increase energy-efficiency [6].

II. A programming model for unreliable hardware

To illustrate the idea of programming model support for unreliable hardware, we describe a few key elements of the design of the Global View Resilience (GVR) system.

A. Creating Reliable State

All programs transform data, so we introduce as a central element, and the basis for reliability, a variation on traditional arrays. GVR introduces versioned arrays, that persist copies of the array data, creating redundant data representation that enables error checking and repair. Application programs indicate when an array should be *versioned*, preserving its contents in a snapshot. The implementation maintains multiple versions of arrays, using them to allows applications to recover from both immediate and latent errors (those that are detected long after their occurrence) by simple rollback and replacement, or more complex data repair.

B. Error checking and correction

GVR enables applications to express error checking functions which can trigger error handling when state corruptions are detected. These error checks are described as procedures which compute over the arrays. It is important to enable application-level checks as there are often application-semantics based invariants for data structures. For example, in a physical system simulation, conservation of mass or energy (as embodied in particle count). In an operating system, the notion that no two pages are allocated to a single page frame. Or in a fluid dynamics simulation that pressure is always positive.

Errors can also be signaled by the hardware or operating system, enabling applications to use redundant information in the application, or computation semantics to correct the corrupted data. For example, a memory error uncorrectable in hardware (double bit error in a SECDED memory), might be correctable based on redundant pointers in a doubly-linked list. In this case, a memory controller would notify the operating system, which would backmap into the application address space for the memory error. Figure 1 shows an example code with application error checking and recovery.

```
void update_particles(gds_t) resilience_prio(low);
int count_particles(gds_t) resilience_prio(high);

alloc(..., PRIORITY_HIGH, &gds_p);
register_error_handler(gds_p, err_handler);

n_orig = count_particles(gds_p);
while (1) { ...
  version_inc(gds_p, 1); /* record version */
  if (finished) break;
  update_particles(gds_p);
  /* If particles not conserved, error! */
  if (count_particles(gds_p) != n_orig)
    raise_error(gds_p); }

status_t err_handler(gds_t gds, error_t err_desc)
    resilience_prio(high) {
  gds_t gds_latest;
  descriptor_clone(gds, &gds_latest);
  do { /* Find a good version */
    move_to_prev(gds);
  } while (full_check(gds) != OK);
  /* Copy good data to resume */
  get(buff, ..., gds); put(buff, ..., gds_latest);
  move_to_newest(gds);
  resume(gds); return OK; }
```

Fig. 1. GVR application-specific error checking, signaling, and recovery in a particle simulation. The global array (`gds_p`) and each function is annotated with resilience priorities.

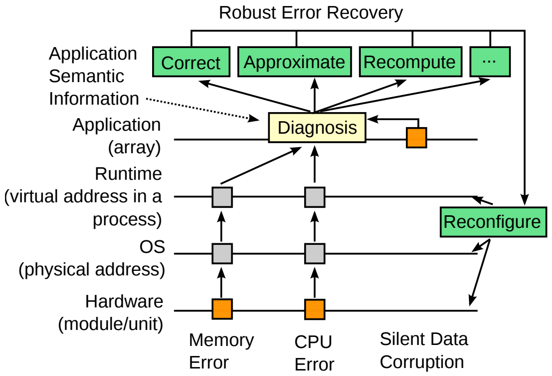

Fig. 2. Unified error handling which enables rich recovery from various errors in an unreliable hardware

C. Resilience priorities

In an unreliable hardware environment, some parts of a program (e.g. error checks and recovery) require highly reliable execution. Amongst the data, certain arrays may be more critical or more difficult to recompute. In GVR, application programmers specify a **resilience priority** for each array, allowing the implementation to optimize the cost for resiliency, without sacrificing the portability.

In the code example (see Figure 1), the code region for updating particle information in the simulation has lower resilience requirements, while the error checking routine has a high priority.

III. Examples of Architecture and OS support

Architecture and operating system support for programming models for unreliable hardware can not only improve execution efficiency, but also increase reliability.

Hardware Architecture

1) report all errors not fully recovered to operating system or runtime, enable recovery at that level
2) mapping chip and DIMM errors to physical address
3) mapping processor errors to affected virtual address space
4) special hardened cores for highly reliable execution, or DVFS switch
5) special hardened memory for highly reliable execution
6) fast NVRAM access for array versioning

Operating System

1) report all errors not fully recovered to application, enable recovery at that level
2) reverse translation (physical → process → virtual) for memory errors
3) identify affected process for processor errors

Runtime/Compiler

1) report all errors not fully recovered to application, enable recovery at that level
2) reverse translation (virtual → application datastructure) for memory errors
3) dynamic computation replication and checking for highly-reliable execution [8]

These examples all assume opening up the error handling for cross-layer error handling as in Figure 2, and support both efficient implementation of primitives but also semantic mapping to enable sophisticated error recovery.

Acknowledgment

This work was supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Department of Energy, under Award DE-SC0008603 and Contract DE-AC02-06CH11357.

References

[1] A. A. Hwang *et al.*, “Cosmic rays don’t strike twice: Understanding the nature of DRAM errors and the implications for system design,” in *Proceedings of ASPLOS ’12*, 2012, pp. 111–122.

[2] V. Sridharan *et al.*, “A study of DRAM failures in the field,” in *Proceedings of SC ’12*, 2012, pp. 76:1–76:11.

[3] D. Henderson *et al.*, “POWER7® system RAS – key aspects of power systems reliability, availability, and serviceability,” http://www-03.ibm.com/systems/power/hardware/whitepapers/ras7.html, October 2012.

[4] Peter Kogge, et. al., “Exascale computing study: Technology challenges in achieving exascale systems,” DARPA IPTO Study Report, available from http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf, 2008.

[5] R. Dreslinski *et al.*, “Near-threshold computing: Reclaiming moore’s law through energy efficient integrated circuits,” *Proceedings of the IEEE*, vol. 98, no. 2, pp. 253 –266, feb. 2010.

[6] M. de Kruijf *et al.*, “Relax: an architectural framework for software recovery of hardware faults,” in *Proceedings of the 37th annual international symposium on Computer architecture*, ser. ISCA ’10, 2010, pp. 497–508.

[7] B. Randell, “System structure for software fault tolerance,” in *Proceedings of the International Conference on Reliable Software*, 1975, pp. 437–449.

[8] G. A. Reis *et al.*, “Swift: Software implemented fault tolerance,” in *International symposium on Code generation and optimization*, ser. CGO ’05, 2005, pp. 243–254.