Towards a Software-Hardware Co-Designed Resilient System

Man-Lap Li, Pradeep Ramachandran, Sarita V. Adve, Vikram S. Adve, Yuanyuan Zhou
Department of Computer Science
University of Illinois at Urbana-Champaign
{manlapli,pramach2,sadve,vadve,yyzhou}@uiuc.edu

***Abstract*— With continued CMOS scaling, future shipped hardware will be increasingly vulnerable to in-the-field faults. To be broadly deployable, the hardware reliability solution must incur low overheads, precluding use of excessive redundancy. We explore a co-designed hardware-software solution that treats most hardware faults as software bugs and leverages common mechanisms for hardware and software reliability, thereby amortizing some of the overhead. Fundamental to such a solution is a characterization of how hardware faults in different microarchitectural structures of a modern processor propagate through the application and OS. In this paper, we first summarize such a characterization for permanent faults. Motivated by this characterization, we discuss our software-hardware co-designed approach for detecting, diagnosing, recovering from, and repairing/reconfiguring around hardware errors.**

I. Introduction

As we move into the late CMOS era, hardware reliability will be a major obstacle to reaping the benefits of increased integration projected by Moore's law. It is expected that components in shipped chips will fail for many reasons including aging or wear-out, infant mortality due to insufficient burn-in, soft errors due to radiation from external sources and the IC package, design defects, aggressive design, process variations, cross-talk, and so on [1]. Such a scenario requires mechanisms to detect, diagnose, recover from, and possibly repair/reconfigure around these failed components so that the system can provide reliable and continuous operation.

The reliability challenge today pervades almost the entire computing market. A reliability solution that can be effectively deployed in the broad market must incur limited area, performance, and power overhead. As an extreme upper bound, the cost of reliable operation cannot exceed the benefits of scaling. A SELSE-2 industry panel converged on a 10% area overhead target to handle all sources of chip errors as a guideline for academic researchers. In this context, traditional high-end solutions involving excessive redundancy are no longer viable. For example, the conventional popular solution of dual modular redundancy for fault detection implies at least a 100% overhead in performance throughput and power. Solutions such as redundant multithreading and its various flavors improve on this, but still incur large overheads in performance and/or power [6].

Two high-level observations motivate our approach. First, in the absence of a "one-size-fits-all" solution with acceptably low-overhead for all markets, a strategy that facilitates a "pay for what you get" approach to the reliability vs. overhead conundrum becomes much more attractive. Second, the hardware reliability solution need handle only the device faults that propagate through higher levels of the system and become observable to software. These software manifestations of hardware faults are analogous to manifestations of software errors in many ways. It may therefore be possible to leverage techniques for handling software errors for hardware faults.

These observations motivate using a unified co-designed hardware/software framework for both hardware and software reliability. Such an approach to hardware reliability could potentially be more cost-effective than software-oblivious approaches because (1) much (but not necessarily all) of the overhead incurred may already be paid for software reliability; (2) a software-aware hardware reliability solution is potentially easier to customize to the target application's requirement of fault coverage vs. overhead (e.g., using application-specific recovery and application-specific tuning of fault coverage vs. overhead); (3) software reliability techniques are potentially more flexible for handling new error sources; and (4) software techniques can support more complex and sophisticated fault diagnoses that are not affordable in hardware-only solutions.

Our co-designed framework consists of the following components: For *error detection*, we use software-level monitoring of error *symptoms* as our primary technique, backed up by hardware-level online testing for further coverage. For *recovery*, we use a combination of hardware- and software-checkpointing and rollback, providing customizable recovery for the application and the OS. For *diagnosis*, we use a common firmware layer to invoke repeated rollback/replay on alternate execution environments to narrow down the source of error, backed up with online hardware test techniques. For *repair and reconfiguration*, we use inherent or explicitly provisioned

This work is supported in part by an IBM faculty partnership award, the MARCO/FCRP Gigascale Systems Research Center, the National Science Foundation under Grant No. NSF CCF-0541383, and an equipment donation from AMD.

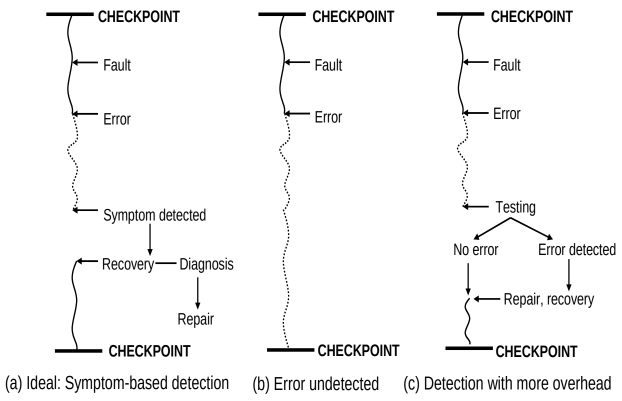

Fig. 1. **Functional overview of the system.**

microarchitecture-level redundancy and reconfigurability.

Fundamental to such a co-designed software-hardware solution is a characterization of how hardware faults propagate to software. We have performed a detailed quantitative characterization for the increasingly important permanent faults to evaluate the feasibility of our proposed approach. This paper summarizes our results and describes our proposed system framework motivated by these results.

II. System Overview

We propose an efficient and tunable framework for achieving hardware reliability, where the hardware gives the system designer control over the degree of reliability. With increasing hardware errors, a "one size fits all" solution that provides near-100% masking will be expensive in terms of area, performance and/or power. A tunable reliability solution facilitates the construction of both high-cost designs, in which nearly 100% of hardware failures are masked by the hardware, and lower-cost designs, where the impact of hardware errors on software is not necessarily almost zero, but is far smaller than the impact of the more frequent software errors.

Figure 1 provides a functional overview of our system. Part (a) shows the best (ideal) case where an error (that will affect software output) and its detected symptom (e.g., application crash) occur within the same checkpoint interval, enabling recovery from the previous checkpoint. The symptom detection invokes a diagnosis procedure that determines if the error is due to a software bug, a transient hardware fault, or a permanent/intermittent hardware fault. This diagnosis procedure may run on a separate co-processor or on another core of a multi-core chip (which is assumed to be fault-free) to detect permanent/intermittent faults in the original core. In such a scenario, the diagnosis procedure isolates the error to the minimal field reconfigurable unit and reconfigures that unit (e.g., switches the unit off, reduces its voltage/frequency, etc.). The system then recovers the software from the previous checkpoint and restarts. In a multicore system, parts of the diagnosis and recovery may occur concurrently.

In a less desirable scenario, the symptom-based detector is unable to detect the error before the next checkpoint (Figure 1(b)). The error may never be detected or may be detected later with a corrupted checkpoint that precludes recovery. Such cases are reduced to a level acceptable for the targeted application by (i) using a collection of low-overhead symptom-based detection techniques that closely watch for deviant software behavior, (ii) using weaker but acceptable notions of recovery that enable larger software checkpoint intervals and hence longer detection latencies, and (iii) providing alternate detection methods that complement symptom-based detection for the more demanding applications and possibly for a subset of microarchitectural structures. Figure 1(c) illustrates the last option – for applications with high reliability demands, we invoke a suite of tests that are performed before the next checkpoint. On a test failure, the system performs diagnosis and recovery as before. The tests can be customized for the desired trade-off in reliability, performance, and power.

III. Quantitative Results

Fundamental to our approach is an understanding of how faults in different hardware structures of a modern superscalar processor propagate through software, and low-cost methods to intercept this propagation within "reasonable" time. We next summarize a quantitative characterization of this propagation for permanent hardware faults (detailed results and methodology are available in [4].)

We focused on permanent faults (vs. transients) because of the increasing importance of such faults due to phenomena such as wear-out and insufficient burn-in, because transients have already been the subject of much recent study, and because permanent faults pose significant challenges different from transients. For example, a permanent fault may manifest to software faster than a transient (because it lasts longer), but for the same reason, it is also less likely to be masked and more likely to corrupt the operating system resulting in an irrecoverable system crash (unless intercepted quickly). Further, after software exposes a permanent fault, the system must diagnose the faulty unit and repair it or reconfigure around it. This is generally expensive, and means that permanent fault detectors cannot afford many false positives (unlike some techniques proposed for transient fault detection [8]).

For our experiments, we used a full system microarchitecture-level simulator that models a modern out-of-order superscalar processor in detail and runs the Solaris-9 Operating System with the UltraSparc-III ISA. The full-system simulation framework allows us to model all system calls and OS interactions of the applications and to understand the effects of permanent faults on the OS as well. We ran SPEC applications

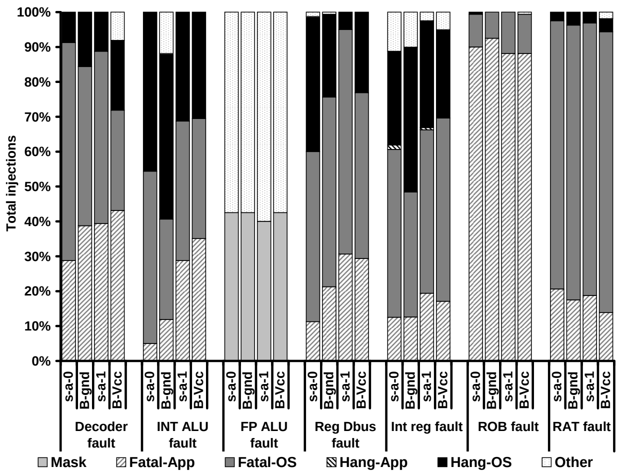

Fig. 2. **Propagation of permanent hardware faults through software. For each microarchitectural structure and fault model, the corresponding bar shows the percentage of injected faults that are architecturally masked and that cause crashes and hangs in the application and OS.**

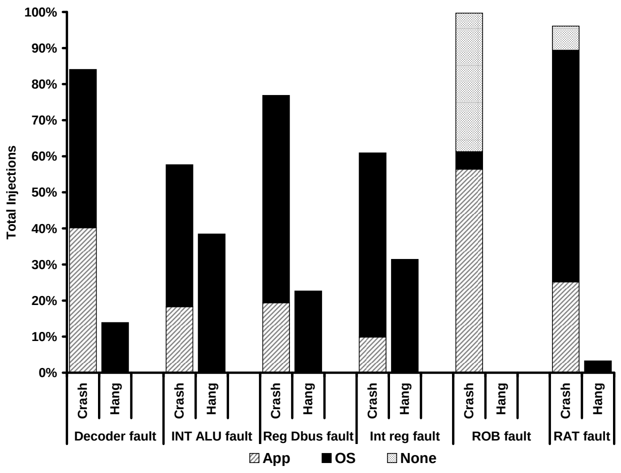

Fig. 3. **For each microarchitecture structure, the figure shows the percentage of crashes and hangs where the architectural state of only the application was corrupted, where the OS (and possibly the application) was corrupted, and where neither were corrupted. The height of each bar gives the percentage of crashes and hangs from faults injected in that structure.**

(on Solaris) on this infrastructure. We injected a total of 4480 permanent faults into several microarchitectural structures of our simulated processor using the stuck-at and bridging fault models. For each fault, we ran the system for 10 million cycles.

For this initial study, we investigated two symptoms of errors: (1) fatal hardware traps that result in application *crashes* and are not typically thrown in correct executions (these are effectively *zero overhead* detection mechanisms), and (2) *hangs* or small infinite loops – a small amount of hardware support can detect these and raise a trap [4]. In our framework, the above traps first invoke a firmware layer that contains the diagnosis and recovery modules (Section IV).

Figure 2 shows the coverage for the above detection mechanisms. Each bar corresponds to one of seven microarchitectural structures and one of the four fault models (stuck-at-0 and 1, and dominant-0 and dominant-1 bridging faults, represented in the figure by B-gnd and B-Vcc respectively), and represents 160 runs (each run injects a fault into a different randomly chosen point in a different bit in the structure in various SPEC benchmarks).

In each bar, the lowest stack represents the percentage of injections that are architecturally masked; i.e., where the fault does not corrupt any architectural state. The next two stacks represent the injections that resulted in a crash of the application (Crash-App) and the OS (Crash-OS) respectively. An injection is said to result in an OS crash if the offending instruction, which causes a fatal trap, is a privileged instruction. The next two stacks represent the injections that result in hangs in the application (Hang-App) and the OS (Hang-OS) respectively. These stacks represent the injections that can be detected using our symptom based detection. The topmost stack (Other) gives the remaining injections – some of these faults could result in crashes or hangs after the observed 10 million instruction period, or they could be masked by the application, or they could result in silent data corruption.

The figure shows that our simple detection mechanisms of crashes and hangs are effective in detecting permanent faults in many microarchitectural structures of a modern processor (even though most such faults are not masked like transient faults). Specifically, over 98% of the faults in all but two structures (INT register and FP ALU) and 93% of the faults in one (INT register) of the remaining two result in crashes and hangs.

On the other hand, faults in some structures like the FP ALU cannot be detected by this method and warrant other software symptoms or alternative hardware support for detection.

Figure 2 also shows that a large fraction of the faults result in crashes and hangs in the OS despite the low OS activity ($<$ 8%) in the fault-free runs. We investigate further and find that the majority of the crashes and all the hangs cause architecture state corruptions in the OS (shown in Figure 3). Therefore, much care is needed to make our system recoverable from OS failures. Unfortunately, current software system practices rarely support checkpointing of the OS.

We therefore investigate the use of hardware for checkpointing OS state. A key aspect that determines the feasibility of using hardware to buffer OS state is the number of instructions from the time the OS architectural state is corrupted until the OS crash or hang. This is presented in Figure 4.

The figure shows that for the microarchitectural structures considered for high-level detection, the number of OS instructions executed from the time that the OS state is corrupted to the time of a crash or hang is

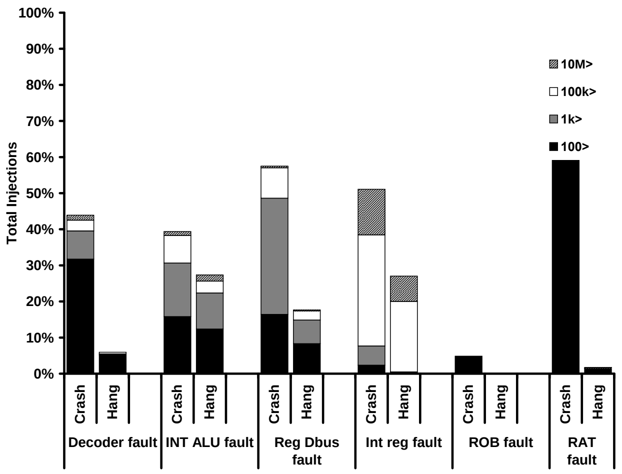

Fig. 4. **Crash and hang detection latency for OS state corruptions in terms of OS instructions executed from the OS architecture state corruption to a crash or a hang, for faults in different microarchitectural components.**

usually small; many of the faults can be detected within 1k instructions. Except for the register file, most faults are detectable within 100k OS instructions. With these detection latencies, hardware recovery techniques such as ReVive [5] and SafetyNet [7] can be applied for full system recovery. Furthermore, our detailed results show that for the longer latency faults, there are few crossings between the OS and application. These results suggest that hardware checkpoint/replay is feasible for the OS, in terms of hardware state required, performance overhead, and simple solutions to the input and output commit problems. We expect that our future work on more sophisticated symptom-based detection techniques will bring the latencies down even further (and also improve coverage).

(Our detailed results also show detection latencies for the application architectural state corruptions [4].)

IV. Resilient System Design

Motivated by the above results, we propose a flexible resilient system framework as follows.

A. Detection

Results discussed in Section III unequivocally show that for several microarchitectural structures, virtually all permanent faults are detected through crashes and hangs. Crashes provide a zero-cost detection mechanism. For hangs, it suffices to invoke a (hardware) hang detector only in the OS. This hardware can initially be a very simple heuristic (e.g., backward branch frequency), which can be further refined in the rare case that it is invoked.

Our results also show that for the floating point unit, alternative mechanisms are required (e.g., space/time redundancy), but these would potentially be much lower overhead than techniques such as full core redundancy. Further, even for the microarchitecture structures that show excellent coverage, there are still some faults that are not detected through crashes and hangs. For these cases, for applications/systems that require higher coverage, we plan to pursue other software level detection schemes (e.g., invariant violation detection, etc.) that can leverage software reliability techniques and address classes of data integrity errors that traditionally result in silent data corruption [9]. For mission-critical systems that require the highest coverage, we plan to explore limited on-line testing to backup our high-level detection.

So far, we have assumed stuck-at and bridging fault models. For future work, we will also investigate how transient and intermittent faults including timing faults propagate from microarchitectural structures to software. Recent work by Gu et al. provides indication of promising results for these fault models as well [2].

B. Diagnosis & Repair

Diagnosis is responsible for determining whether a detected symptom is due to a software bug, transient hardware fault, or a permanent/intermittent hardware fault. For the last case, diagnostic tests are needed to isolate the fault to the minimum field reconfigurable unit.

Our diagnosis process exploits the hardware- and software-checkpointing and rollback recovery mechanisms in the framework (described in the next section). During diagnosis, the execution is repeatedly replayed from its checkpoint both on the same and a different core (assuming a multicore environment) so as to expose the fault type.

For example, the first step of the diagnosis procedure is to use the hardware recovery mechanism to replay the execution from the previous hardware checkpoint. If no error symptoms are triggered, then we conclude the original error was transient and contained within the hardware recovery window. Otherwise, it is possible that the checkpoint was corrupted due to an undetected fault before the checkpoint, or the hardware checkpoint was correct but the hardware has a permanent fault that prevents successful replay. The diagnosis procedure cannot distinguish these two cases yet. It therefore invokes a replay of the execution on another core (assuming a multicore environment). Depending on whether that replay produces an error symptom, the diagnostic procedure makes further deductions.

Depending on the granularity desired, if a permanent/intermittent hardware fault is found, the diagnosis may probe further within a processor and rely on hardware-level online testing techniques to locate the fault.

Repair/reconfiguration is relevant to permanent faults and interacts with detection and diagnosis. We rely on built-in redundancy within current processors and optional sparing (at the level of a core and at the level of individual microarchitectural units), as well as voltage/frequency scaling for timing error repair. The appropriate granularity of repair depends on the effectiveness of detection and diagnosis.

C. Recovery

Our framework uses both hardware-based and software-based recovery strategies as well as hybrid recovery to recover both the application and the OS. Each recovery strategy has different cost and overhead as well as different recovery window lengths and capabilities. The reason for involving different recovery strategies is to enable customization for systems with different reliability demands.

A specific challenge is the recoverability of the OS. Our results show that a large fraction of the faults result in corrupting the OS; therefore, much care is needed to make our system recoverable from OS failures. At the same time, we find that there are usually only a relatively small number of OS instructions executed between an OS state corruption and a crash/hang and longer latency faults often have few crossings between the OS and application. These results suggest that hardware checkpoint/replay techniques, such as ReVive [5] and SafetyNet [7], with a relatively small amount of state are feasible for recovering the OS. In the future, we will perform more extensive experiments with more OS intensive applications to test this claim, and determine if alternative techniques that require invasive OS restructuring for resilience are needed.

For application recovery, we find that the detection latencies are also often within the hardware recovery window; however, there are also many cases of higher latency. The high latency cases can be handled using software checkpointing, with an application specific trade-off between buffering persistent outputs/inputs for a few milliseconds of the execution and full application recovery.

D. Implementation

Based on the above discussion, we summarize here the key implementation components assumed by our framework.

- *Multicore environment or reliable coprocessor:* Diagnosis and recovery assume a reliable coprocessor or another fault-free core on chip.
- *Hardware Test Library (HTL):* This is a thin layer of firmware used for diagnosis and test-based detection. This layer provides the API to selectively control the different reconfigurable components of the processor.
- *R-visor:* This layer of software runs below the OS and is responsible for coordinating error diagnosis, recovery, and repair through the use of the HTL. The R-visor is required to run reliably on an alternative core or on a simple (reliable) coprocessor, but is invoked very rarely on detection of an error or for periodic testing (if supported). The R-visor layer is analogous to the Itanium Machine Check Architecture firmware layer that facilitates software assisted recovery after hardware error detection [3].
- *Hardware, software checkpoint/replay mechanisms:* Hardware checkpoint/replay mechanisms are essential for efficient recovery of both the OS and the application. However, for errors that have long detection latency, software checkpoint/replay mechanism is needed for full application recovery.
- *Reconfigurable hardware:* For any resilient hardware, some reconfigurability is required to reconfigure the defective components.
- *Hardware hang detector and other future symptom detectors:* A low-overhead hardware hang detector is required for detecting both application and system hangs. In the future, we also expect to use some hardware support for more sophisticated software level error symptom detectors.

V. Acknowledgments

We thank Pradip Bose for many discussions throughout the study and Subhasish Mitra for many invaluable comments on the paper.

References

[1] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, 2005.

[2] W. Gu, Z. Kalbarczyk, and R. K. Iyer, "Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2004, p. 887.

[3] *Itanium Processor Family Error Handling Guide*, Intel Corporation, April 2004, document no.: 249278-003.

[4] M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou, "Understanding the propagation of hard faults to software and its implications on resilient systems design," UIUC CS, Tech. Rep. UIUCDCS-R-2007-2822, February 2007.

[5] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors," in *Proceedings of 29th International Symposium on Computer Archritecture*, 2002.

[6] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," in *FTCS '99: Proc. of Twenty-Ninth Intl. Symp. on Fault-Tolerant Computing*, 1999.

[7] D. Sorin, M. M. Martin, M. Hill, and D. Wood, "SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *Proceedings of 29th International Symposium on Computer Archritecture*, 2002.

[8] N. Wang and S. Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors," *Dependable and Secure Computing, IEEE Trans. on*, vol. 3, no. 3, July-Sept 2006.

[9] P. Zhou *et al.*, "AccMon: Automatically Detecting Memory-related Bugs via Program Counter-based Invariants," in *37th IEEE/ACM Intl. Symp. on Micro-architecture (MICRO)*, 2004.