Architectures for Online Error Detection and Recovery in Multicore Processors

Dimitris Gizopoulos
Mihalis Psarakis
Dept. of Informatics
University of Piraeus, Greece
{dgizop|mpsarak}@unipi.gr

Sarita V. Adve
Pradeep Ramachandran
Siva Kumar Sastry Hari
Dept. of Computer Science
University of Illinois at Urbana-Champaign, USA
swat@cs.uiuc.edu

Daniel Sorin
Albert Meixner
Dept. of ECE and Computer Science
Duke University, USA
sorin@ee.duke.edu
albert.meixner@gmail.com

Arijit Biswas
TRU Group
Intel Corporation, USA
arijit.biswas@intel.com

Xavier Vera
Intel Barcelona Research Center
Intel Labs Barcelona – UPC
xavier.vera@intel.com

Abstract— **The huge investment in the design and production of multicore processors may be put at risk because the emerging highly miniaturized but unreliable fabrication technologies will impose significant barriers to the life-long reliable operation of future chips. Extremely complex, massively parallel, multi-core processor chips fabricated in these technologies will become more vulnerable to: (a) environmental disturbances that produce transient (or soft) errors, (b) latent manufacturing defects as well as aging/wearout phenomena that produce permanent (or hard) errors, and (c) verification inefficiencies that allow important design bugs to escape in the system. In an effort to cope with these reliability threats, several research teams have recently proposed multicore processor architectures that provide low-cost dependability guarantees against hardware errors and design bugs. This paper focuses on dependable multicore processor architectures that integrate solutions for online error detection, diagnosis, recovery, and repair during field operation. It discusses taxonomy of representative approaches and presents a qualitative comparison based on: hardware cost, performance overhead, types of faults detected, and detection latency. It also describes in more detail three recently proposed effective architectural approaches: a software-anomaly detection technique (SWAT), a dynamic verification technique (Argus), and a core salvaging methodology.**

Keywords: multicore microprocessors; dependable architectures; online error detection/recovery/repair.

I. INTRODUCTION

Dimitris Gizopoulos

Advances in semiconductor manufacturing processes have sustained the validity of Moore's law for several decades both in terms of device counts and delivered performance. Recently, a consensus has been reached in the computing community that the only viable way to keep performance improvement rates within a given power budget is by building *multicore processors* and exploiting massive parallelism. This major paradigm shift in computing comes with several challenges affecting all aspects of hardware and software technologies: circuit design and manufacturing, microprocessor architectures, memory systems, programming languages, compilers, and operating systems.

A major challenge which is now more important than ever before in the history of computing is *dependability* [1]. Traditionally high dependability/reliability was mandatory only for a few applications and systems where cost was not a major limitation. Multicore microprocessors (and memories) are today manufactured in inherently unreliable technologies. *Cost-effective dependability* for general purpose computing systems is now a demand: dependability on multicore chips and computing systems built with unreliable components requires a synergy of effective hardware and software solutions.

The most important sources of unreliable hardware operation that can lead to system failures are ([2]): (i) process variability that causes the heterogeneous operation of identical components on the same chip; (ii) soft/transient errors sensitivity of today's very deep submicron circuits; and (iii) accelerated aging/wearout of devices due to their extreme operating conditions. In addition to the problems of hardware errors that can severely affect the correct operation of multicore microprocessors, there is another major threat that continues to worsen. Due to the extreme complexity of multicore processors and the pressure for reduced time-to-market, even after the application of a comprehensive pre-silicon verification and post-silicon validation flow, major design errors/bugs can still exist after the chip enters operation in the field. It is apparent that success of the emerging multicore microprocessor paradigm depends (among many factors) on the effective deployment of online error detection, recovery and repair schemes that can provide low-cost dependability guarantees against hardware errors and design bugs.

In this paper, we summarize of few of the most representative error detection and repair techniques that have been proposed in the literature for the building of dependable multicore architectures. We provide taxonomy of the error detection approaches and compare them based on various criteria: hardware and performance overheads, detection latency, targeted faults and fault coverage. The rest of the paper discusses three such architectural approaches in more detail.

II. DEPENDABLE MULTICORE PROCESSOR ARCHITECTURES

Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera

A. *Online Error Detection*

Multicore processor architectures incorporate CPU cores, memory arrays (e.g. caches, register files), memory control logic and interconnection logic. Memories that occupy a large portion of processor die can be successfully protected using well-known information-redundancy techniques like error-correcting codes (ECC). Thus, the key element of online error detection is to protect the remainder of the processor: the CPU cores (which dominate the remaining die area), the memory hierarchy control logic (memory consistency), and the interconnection logic. Several online error detection techniques for the aforementioned processor components have been recently proposed. These approaches can be classified in four main categories as shown in Fig.1: (a) *redundant execution* approaches which exploit the inherent replication of processor cores and threads in a multicore processor architecture, (b) *periodic built-in self-test (BIST)* approaches which advocate leveraging the built-in test mechanisms of processors traditionally used for manufacturing testing, (c) *dynamic verification approaches*, and (d) *anomaly detection approaches*.

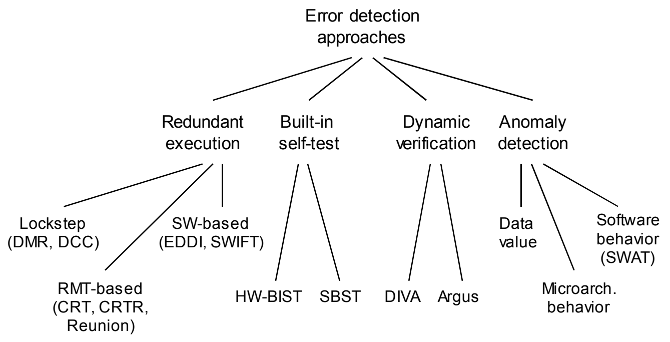


Figure 1. Taxonomy of online error detection techniques.

Redundant execution. In a redundant execution approach two independent threads execute copies of the same program and results are compared. With the advent of multiple on-chip threads in simultaneous multithreading (SMT) and chip multiprocessor (CMP) architectures, hardware redundancy techniques such as dual modular redundancy (DMR) and triple modular redundancy (TMR) – which have been studied for a long time but impose very high hardware overheads – have become more attractive; keep in mind that full utilization of cores is not usually feasible and, therefore unused processor cores can execute redundant threads. The two dominant forms of redundant execution in processor architectures are the *lockstep configuration* and the *redundant multithreading (RMT)* with loose lockstepping or without it [3]. In a typical lockstep configuration, identical cores are tightly-coupled in a per cycle or per instruction basis. Aggarwal et al. [4] propose DMR and TMR configurations for CMPs which provide error detection and error recovery through fault containment and component retirement. The proposed technique needs a small amount of extra area to support the reconfiguration-for-repair mechanism (less than 1% in a commodity processor). LaFrieda et al. [5] present a dynamic core coupling (DCC) technique for CMPs which allows arbitrary processor cores to verify each other in a DMR setup avoiding static binding of cores.

Mukherjee et al. [6] propose a redundant execution technique – named chip-level redundant threading (CRT) – that extends RMT technique for single SMT processors to CMP architectures. Similar to RMT, CRT uses loosely synchronized redundant threads reducing the checker overhead. A leading thread in one core is checked by a trailing thread in another core forwarding their results through a dedicated bus. Application on a dual-core SMT processor showed that CRT achieves better results than simple lockstepping the two cores. Also, it achieves better permanent fault coverage than RMT techniques since the redundant threads do not share common resources. Gomaa et al. [7] propose Chip-level Redundantly Threaded multiprocessor with Recovery (CRTR) which extends CRT for transient-fault detection in CMPs. CRTR uses a long slack enabled by asymmetric commit to hide inter-processor latency required in CRT. Smolens et al. [8] propose Reunion, a CRT-based architecture that relaxes input replication while preserving the existing memory system, including the coherence protocol and consistency model and reduces comparison bandwidth by compressing the results.

Unlike hardware-based redundant techniques which impose significant hardware overhead, *software-based redundant techniques* can provide a low-cost alternative. Oh et al. [9] proposed EDDI (Error Detection by Duplicated Instruction), a software-based error detection technique in which all instructions are duplicated and appropriate instructions are inserted to check the results. Although more than 100% performance overhead is expected due to instruction duplication, in most cases it is less than 100%; the lower overhead is because the duplicated programs do not have dependencies and thus, the shadow program may fill the empty slot of the pipeline. SWIFT [10] makes several key refinements to EDDI; the major difference is that while the sphere of replication (the domain of redundant execution) in EDDI includes the memory subsystem, SWIFT leaves it out, assuming that the memories are protected by a well-established ECC technique. EDDI and SWIFT are single-threaded approaches that can be applicable in both unicore and multicore processors.

Periodic built-in self-test. The abovementioned redundant execution approaches support *concurrent* error detection because redundant hardware (or software) runs concurrently with the normal one. Another category of error detection approaches leverages the use of built-in self-test (BIST) mechanisms (hardware or software) traditionally used for manufacturing testing. The BIST-based approaches perform *non-concurrent* error detection because the self-test sessions are executed either periodically or during idle time intervals. Hardware BIST techniques [11] are well-established DFT solutions which increase the system testability and relaxes tester's interface speed requirements during manufacturing testing. Shyam et al. [12] utilize existing distributed hardware BIST mechanisms to validate the integrity of the processor components in an online detection strategy. The proposed error detection technique provides high fault coverage (89%) imposing low area overhead (5.8%). Software-based self-test (SBST) has gained increasing acceptance for microprocessor testing the last years and currently forms an integral part of the processor manufacturing test flow [13]. The key idea of SBST is to exploit on-chip programmable resources to execute normal programs that test the processor. Functional test patterns are generated and applied by the processor using its native instruction set, virtually eliminating the need for additional test-specific hardware while the test is applied at the actual operating frequency. SBST has been recently exploited in multicore and multithreaded architectures. Apostolakis et al. [14] apply SBST to bus-based CMPs and propose a test scheduling methodology to exploit core-level execution parallelism and reduce the total test execution time. Foutris et al. [15] extended the SBST methodology of [14] to multithreaded CMP architectures. The proposed methodology speeds up test execution by exploiting execution parallelism and simultaneously increases the fault coverage (88%). Constantinides et al. [16] propose an error detection methodology using periodic execution of SBST tests, assisted by ISA extensions and microarchitectural support. Application of software tests implies a system overhead since the periodic testing time may vary between 5% and 25% of the system time.

Dynamic verification. Another error detection category that does not use redundant execution is *dynamic verification*. These approaches operate at runtime and use dedicated hardware checkers to verify the validity of specific *invariants* assumed to be true in error-free operation. The key point in a dynamic verification approach is to define a comprehensive set of invariants. Dynamic verification was first introduced in dynamic implementation verification architecture (DIVA) [17]. DIVA uses a simple checker core to detect errors in a speculative, superscalar core. DIVA is an excellent low-cost solution for complex superscalar processors where the checker imposes a relatively small area overhead (6% for an Alpha 21264 processor [18]). However, for simpler processors typically used in multicore architectures the complexity of checker core is comparable with that of processor core and it significantly increases the cost. A more recent dynamic verification approach, Argus [19], checks four invariants: control flow, computation, dataflow and memory integrating existing checking mechanisms. Argus architecture imposes less than 17% area overhead to a RISC processor core while still achieves high fault coverage (98%). More details about dynamic verification and especially Argus architecture are discussed in Section IV.

Dynamic verification approaches are also used to validate the cache coherence [20], [21], and the memory consistency [22], [23], [24] of the memory hierarchy system of multicore processor architectures for either online error detection or post-silicon validation. Meixner and Sorin [20], [24] implement low-cost checkers to verify various invariants that a specific memory consistency model must satisfy in error-free operation. Pascual et al. [21] extends a cache coherence protocol to deal with transient faults that affect the interconnection network of a CMP. Chen et al. [22] captures the ordering of shared-memory operations and periodically validates the ordering using a constraint graph. DeOrio et al.,[23] log memory operations in on-chip storage resources and periodically aggregate and check the logs to validate memory consistency in post-silicon validation.

Anomaly detection. These approaches detect faults monitoring the software for *anomalous behavior*, or *symptoms* of faults, using low-cost hardware and software monitors. The anomaly detection approaches can be classified in three categories according to the level of the symptoms they detect ([3]): (a) those that detect data value anomalies [25], like out-of-range values, values not matching with value history, bit invariants, etc., (b) those that detect microarchitectural behavior anomalies [26] like exceptions, cache misses, page faults, etc. and (c) those that detect software behavior anomalies [27], like fatal hardware traps, abnormal application exit, OS hangs, etc. Software behavior anomalies approaches based on SWAT (SoftWare Anomaly Treatment) architecture are discussed in Section III.

Table I compares all the different error detection categories in terms of hardware cost, the extra hardware (if any) required, performance overhead, i.e. overhead imposed to the system due to the additional time for error detection, detection latency, i.e. the time between error appearance and error detection, targeted faults, i.e. the fault types detected by the technique and fault coverage, i.e. the percentage of detected faults.

TABLE I. COMPARISON OF ONLINE ERROR DETECTION TECHNIQUES

Note: DMR occupies twice the number of cores; thus imposes 100% hardware cost (*) and reduces the effective number of cores by one half (**).

Error detection technique	Hardware cost	Performance overhead	Detection latency	Targeted faults	Fault coverage
Lockstep redundant execution	DMR [4]: < 1% for reconfiguration support DCC [5]: 64-entry age table in each core to support master-slave consistency (*)	(**) DCC: 3%-5%	Cycle-by-cycle lockstep	Transient and permanent faults	
RMT redundant execution	CRT [6], CRTR [7], and Reunion [8] require extra hardware: queues, inter-processor communication and checker module	CRT: achieves 13% better performance than a dual lockstep CPU Reunion [8]: 5%-6%	Loose locktep: tens of cycles (due to interprocessor communication and checker latency)	Transient and permanent faults	Better permanent FC than single RMT
Software-based redundant execution	EDDI [9], SWIFT [10]: None	EDDI [9]: Less than 100% in most cases	Low	Transient faults	EDDI: 98.5% SEU
Built-in self-test (BIST)	HW-BIST [12]: 5.8% SBST [13], [14], [15]: None	Depends on the self-test execution frequency [17]: 5%-25%	Test period (maximum)	Permanent faults	[12]: 89% [14]: 91% [15]: 88%
Dynamic verification	DIVA [17]: 6% in a superscalar processor. Much higher in simpler cores Argus [19]: 17% in a RISC core	Low Argus [19]: 3.2-3.9%	Low	Transient and permanent faults and design bugs	Argus: 98% (smaller than DIVA)
Anomaly detection	SWAT [27]: low	Low	95% within 100K cycles 98% within 10M cycles	Transient and permanents faults	> 99%

B. *Online Error Recovery and Repair*

Error recovery techniques are classified into two broad categories: forward error recovery (FER) and backward error recovery (BER). FER techniques detect and correct the errors without requiring to rollback to a previous correct state. This can be achieved only using redundancy, e.g. a TMR lockstep configuration. Backward Error Recovery (BER) techniques periodically save (checkpoint) system state and rollback to the latest validated checkpoint when a fault is detected. Efficient checkpoint and rollback approaches have been proposed in the literature for multicore architectures [28], [29], [30]. These approaches can be classified based on three characteristics: the sphere of BER (register files, caches, memory), the relative checkpoint location (dual or leveled) and the separation of checkpoint and active data (full or partial). *SafetyNet* [28] combines local checkpointing and incremental logging of data and stores updates in special buffers (dual, partial separation by logging). *Revive* [29] uses global checkpointing, flushes cache dirty lines to memory and uses a special directory controller to log memory updates in memory (dual, partial separation by logging). SafetyNet can tolerate fault detection latency up to 1 ms while Revive up to 100 ms. Revive I/O [30] is an extension of Revive that deals with the output-commit problem.

Error repair techniques typically leverage redundancy (either spatial or temporal) to deactivate and isolate the faulty component from the rest of the system. Different approaches [31], [32], [33] have proposed reconfiguration and repair techniques for complex, superscalar processor architectures based on the fact that such architectures include inherent redundancy to provide increased performance and speculative execution. Thus, the redundant and non-essential components could be disabled in order to improve yield and enable graceful performance degradation of the system.

However, the redundant functionality of complex superscalar cores is not usually included in the processing cores of a multicore architecture and therefore the latter does not allow the application of the above repair approaches. Meixner and Sorin [34] proposed *Detouring*, a software-based error repair technique for simple cores which are more attractive for building massively parallel multicore architectures. Given that simple cores do not have sufficient redundancy, the key idea of Detouring is to provide fault tolerance by software: software is modified in order to preserve its functionality but not to use the faulty components. Detours have been proposed for several components, e.g. instruction cache, registers, functional units, etc. Detouring imposes no hardware and performance overhead when the cores are fault-free.

Recent approaches [4], [35]-[38], propose repair techniques allowing reconfiguration for multicore processor architectures. The effectiveness of these techniques should rest on the coordination of fault diagnosis and isolation at several levels, i.e., at circuit level (fine granularity) or at architectural level (coarse granularity) and the allocation of processing threads to fault-free components. Approaches [4] and [35] adopt the straightforward architectural level solution, i.e. the faulty core is deactivated and replaced by another spare core. Some approaches consider the repair of interconnection network. Collet et al. [35] propose a self-organizing approach that -tests and mutually diagnoses the CPUs, routers, and on-chip memories in a multicore array, isolates and deactivates the defective elements and discovers valid routes.

Recent approaches [36]-[38] consider reconfiguration at finer granularity to reduce performance degradation in high failure rates. *StageNet* fabric proposed by Gupta et al. [36], is a reconfigurable multicore architecture which is designed as a reconfigurable network of processor pipeline stages rather than isolated cores. The pipeline stages act as processing elements and are shared among cores providing inherent fine-grained redundancy. Romanescu and Sorin [37] propose *Core Cannibalization Architecture*, which allows cannibalization of the cores into spare parts, where these parts can be pipeline stages. Powell et al. [38] propose *architectural core salvaging* based on the observation that a multicore architecture can be ISA-compliant, i.e. executes all instructions of the ISA, even if some defective cores cannot execute it entirely. Exploiting cross-core redundancy, the victim thread can migrate to another core that can execute the required operations. Core salvaging approach is presented in Section V.

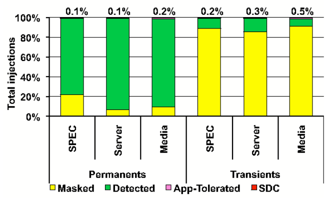
III. SWAT: Designing Resilient Hardware by Treating Software Anomalies

Pradeep Ramachandran, Siva Kumar Sastry Hari, Sarita V. Adve

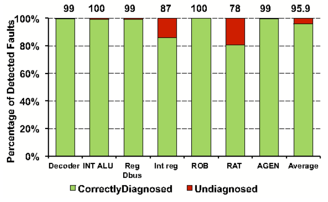
SWAT (SoftWare Anomaly Treatment) is a comprehensive solution to detect, diagnose, and recover from a variety of hardware faults at very low cost. SWAT is based on two key observations. First, a reliable system must handle only those hardware faults that propagate to software and cause anomalous behavior; the rest can be safely ignored, lowering the incurred cost. Second, despite the impending reliability threat, fault-free operations remain common and must be optimized. SWAT thus detects hardware faults by watching for anomalous software behavior, using zero to low-cost hardware and software monitors. In the rare event of a fault, SWAT invokes a comprehensive diagnosis procedure that isolates the source of the fault in a multicore system, facilitating fine-grained repair or reconfiguration. SWAT performs recovery by using a checkpointing and rollback based mechanism and supports recovery even in the presence of I/O.

A. SWAT Components

Fault Detection: SWAT detects hardware faults by monitoring anomalous software execution (symptoms). These symptom detectors incur near zero overhead in fault free execution. SWAT deploys the following detectors: (1) *Fatal Traps* indicating an illegal software operation, e.g. a divide by zero, (2) *Hangs* indicating an application or system hang, identified with a heuristic hardware hang detector, (3) *Kernel Panics* indicating that the kernel crashed due to a fault, (4) *High OS* indicating an anomalously high amount of contiguous OS activity, (5) *Application Aborts* indicating an application that was terminated due to illegal operations, (6) *Out-of-bounds Detector* flagging loads/stores to addresses outside legal bounds. These detectors can be implemented with existing hardware performance counters and minimal hardware support. Hence, they incur near-zero performance and area overheads. Fig. 2(a) shows their efficacy to detect permanent and transient faults in the core for a variety of workloads. The low rate of Silent Data Corruptions (SDCs, numbers on top of each bar) shows that these detectors are highly effective in single-core [27] and multicore systems [39] (SDC rate between 0.1% and 0.5%). We have also explored the use of software-level program invariants, extracted by using a compiler, as detectors of hardware faults in the iSWAT framework [40]. Our results show that such detectors may be used to further reduce the low SDC rates, demonstrating the customizability of SWAT to the system needs.



(a) Efficacy of detectors



(b) Multicore fault diagnosis

Figure 2. Efficacy of SWAT to (a) Detect and (b) Diagnose in-core faults.

Fault Diagnosis: After a symptom is detected, the SWAT diagnosis module takes over to identify the root-cause of the symptom assisted by the recovery module. Simple rollback and re-execution is sufficient to diagnose a transient fault and even recover from it. If the fault is not diagnosed as a transient fault, it can be either a software bug or a permanent hardware fault. To diagnose a permanent hardware fault, SWAT performs a series of rollback and re-executions to compare the traces and isolate the faulty core in a multicore system. If no permanent fault is diagnosed and all the re-executions detect the same SWAT symptom then a software bug is inferred. For permanent hardware faults, SWAT first identifies the faulty core and then proceeds with the fine grained microarchitecture level in-core fault diagnosis.

Multicore fault diagnosis: Multithreaded applications running on multicore systems often share data across threads. This makes diagnosis hard because a fault may escape a faulty core and affect a fault-free core. mSWAT [39] diagnoses the faulty core even in the presence of fault propagation across cores by addressing the following two key challenges: reducing the high cost involved with deterministic replay of a multi-threaded execution, and eliminating the requirement for fault-free spares core for diagnosis. It tackles these challenges by devising a light-weight replay technique that can deterministically replay the execution of each thread in isolation of the other threads. It then uses the isolated deterministic replay to synthesize an inexpensive selective TMR execution only for the purpose of diagnosis. mSWAT successfully diagnoses a large fraction of detected faults. Fig. 2(b) shows the diagnosability of detected faults in various microarchitecture units while running multithreaded media workloads. Over 95% of the faults are successfully diagnosed; all faults escaping to fault-free core are successfully diagnosed.

Single-core fault diagnosis: In most modern systems disabling an entire complex core is wasteful. SWAT therefore proposed a microarchitectural diagnosis procedure called Trace Based Fault Diagnosis (TBFD [41]) that refines the diagnosis further to the microarchitecture level to exploit the built-in microarchitectural redundancy to reconfigure around failed components. TBFD exploits the presence of a fault-free core in the system (identified by mSWAT) to replay and compare the executions on the faulty and fault-free cores. With a sophisticated algorithm that uses violated microarchitecture-level invariants as diagnosis hints, TBFD successfully diagnoses over 98% of the faults to the faulty component, enabling fine-grained repair.

Fault Recovery: Since SWAT, like other symptom detectors, allows the fault to corrupt the architecture state, it relies on checkpointing and rollback support for recovery. While previously proposed hardware checkpointing techniques, such as SafetyNet [28] and ReVive [29], have demonstrated low-cost techniques to recover the architecture state, they have ignored the notorious *output-commit problem*. The only previous recovery solution that handles this problem (ReVive-I/O [30]) relied on software support and could not guarantee that the committed outputs were protected from in-core faults. SWAT uses a low-cost simple hardware buffer for buffering outputs in hardware and circumvents the limitations of existing solutions. Further, such a solution demonstrates that the detectors need to detect the faults in sub-millisecond durations (instruction latencies of $\leq$ 100K instruction) in order to keep overheads on fault-free execution from delaying outputs minimal. Previous work has not identified this constraint because output buffering has largely been ignored. Under such a constraint, our SWAT detectors are effective recovering from over 95% of the permanent and transient hardware faults injected into distributed client-server workloads even in the presence of system I/O while incurring less than 5% performance overhead on fault-free execution.

IV. Dynamic Verification of Cores and Memory systems

Daniel J. Sorin, Albert Meixner

There are many ways to detect errors, but many of them are too costly – in terms of power, energy, performance, or area – to be viable for commodity processors. An attractive, low-cost approach to error detection is to dynamically check that certain system-wide invariants are being maintained, and this process is known as *"dynamic verification"* (or "online testing"). By virtue of checking invariants, rather than checking specific components, dynamic verification is independent of the specific implementation and can detect errors due to soft and hard faults as well as errors due to design bugs. Dynamic verification schemes can achieve excellent error detection coverage at overheads in the 1-15% range.

Dynamic verification seems the obvious solution for error detection, dealing with two important, inter-related challenges. The first is that we must determine what invariants to check. Ideally, we would identify a set of invariants that, if checked, would be sufficient for detecting any error in the system. We may need to "divide and conquer", i.e., subdivide the system into components for which we can identify invariants. The second challenge is implementing efficient checkers. Implementation often requires us to reformulate the invariants in a way that is conducive to being checked by hardware. Implementation constraints may also lead to the checker hardware being probabilistic (e.g., uses lossy checksum).

There is a set of recently developed schemes for dynamically verifying multicore processors and in particular focus on dynamic verification of processor cores and cache-coherent shared memory (interconnection network, coherence, etc.). Different sets of invariants have been proposed to check and specific hardware designs check them at runtime. Viability of the schemes has been confirmed by experimental results.

For detecting errors in processor cores, the Argus [19] approach for error detection has been proposed. The key idea behind Argus is that a von Neumann core performs only three activities: control flow (choosing which instructions to execute), computation (performing the computation for each instruction), and dataflow (passing results from a producer instruction to consumer instructions). By checking each of these activities at runtime, Argus can detect virtually any possible error in the core. Checkers have been implemented for each activity, and experimental results have been presented that confirm that Argus detects errors at low-cost.

For detecting errors in the memory system, dynamic verification of memory consistency (DVMC) [24] has been proposed. Because a memory consistency model defines the correct behavior of the memory system, the model serves as a complete invariant for dynamic verification. That is, by dynamically verifying memory consistency, we can detect all possible errors. The problem is solved by a divide-and-conquer approach by splitting consistency into sub-invariants. The most challenging sub-invariant for dynamic verification is cache coherence, and a scheme for achieving this goal is presented [20].

V. How to Manage Accidentally Heterogeneous Cores

Arijit Biswas

A. *Heterogeneity by Accident*

The search for higher performance with optimal power in the era of multicore CPUs has given rise to the notion of heterogeneous cores. The main concept behind such CPUs is that a general purpose one-size-fits-all core is not necessarily the best for optimal power/performance. In some cases, it may be more effective to have different types of cores on a single CPU die in order to best accommodate various needs of different applications. While such CPUs tend to be more difficult to program due to their asymmetric nature, they are nevertheless gaining in popularity in many computing areas.Heterogeneous cores are becoming popular, but multicore CPUs with symmetric general purpose cores still reign supreme. They are easier to program and provide more consistent performance in many instances. However, such non-heterogeneous CPUs may become heterogeneous, not by design, but as a result of permanent defects that may render a core unable to correctly execute certain instructions.

Multicore CPUs devote a large fraction of die area to regular memory structures, mainly caches. Fortunately, caches can be protected from manufacture-time defects using well-known techniques. Thus, the remainder of the die becomes the major source of defect vulnerability. The bulk of this remainder is CPU cores. In the past, such a defect would cause the entire CPU die to be rejected as defective. Recent work has shown promise that such defects are tolerable if they are properly managed.

Managing defective cores is a two-part challenge: defect detection and defect tolerance. In this section, we focus on solutions for defect tolerance. One obvious solution set, which we define as core disabling and core sparing, disables defective cores (disabling) or enables spare standby cores (sparing) in the event of a core defect. Core disabling reduces sales price due to smaller core count, and core sparing consumes precious die area while providing no performance or economic benefit in a non-defective die.

B. *Core Salvaging*

A more desirable alternative to core disabling and core sparing is core salvaging, which allows defective cores to continue operation. Microarchitectural core salvaging techniques disable defective execution pipelines [32] or schedule operations on alternate or spare resources [42], [31], [33], [37] to avoid utilizing the defective area, suffering performance loss due to defects. Microarchitectural core salvaging exploits microarchitectural redundancy, relying on the ability of a core to execute the entire ISA correctly even in the presence of certain defects. Ideally, the performance impact of defects is minimal, so that the presence of a single core with a tolerated defect on a multicore die is negligible.

Most modern cores contain large amounts of redundant logic which is generally used to improve performance. This redundant logic could potentially be used to compensate for the defective logic if the defect exists in an opportune location to facilitate this. This comes at the cost of being able to test for and isolate the defect to a finer granularity than just a core. It turns out however that such opportunities for taking advantage of this technique are actually quite small and may require significant overhead.

A better solution that offers significantly more benefit than microarchitectural redundancy alone, is called architectural core salvaging [38]. Architectural core salvaging leverages the fact that a single core need not be ISA compatible so long as the CPU as a whole is. What this means is that if a defect on a particular core renders it unable to execute certain instructions, that core is still usable assuming that we can detect and move the un-executable instructions to a different core.

C. Results

Recent work has shown that microarchitectural redundancy does not cover defects as well as previously thought [38]. This is because large portions of many redundant structures such as multi-entry arrays are actually made up of non-redundant logic such as decoders, buffers, and interconnect which is not covered. Other structures like execution units are often used by multiple instructions, some of which only use a particular unit even though redundant functionality exists. Microarchitectural techniques have been shown to cover only ~10% of the non-cache core area.

This work also shows that architectural redundancy, the same redundancy exploited by architectural core salvaging, can cover significantly more area. While there are always defects that will render a core useless, such as the inability to execute memory operations, most ISAs contain numerous instructions which are used infrequently yet the hardware dedicated to them occupies significant area. A good example includes SIMD instructions. Not all applications include these instructions in their mix. Even many applications that do have them only have them in specific portions of the code. In such cases, it would be possible to trap on any such instruction and migrate the thread to a "good" core.

Further, it may be possible that better thread-scheduling and thread-swapping algorithms may be able to better harness and use this technique. Even with simple thread-swapping algorithms, architectural core salvaging has been shown to cover nearly half the execution units on an IA32-like processor.

Powell et al. [38] also shows that architectural core salvaging becomes compelling as the number of cores on a CMP increases. Further, this technique is orthogonal to core sparing, which would continue to be used if a major defect compromises a core such that even basic instructions could not execute correctly. Combining microarchitectural and architectural core salvaging can cover significant portions of the processor core while gaining nearly the full performance out of a core with minor defects. Powell et al. shows a 21% coverage for protecting only a small handful of structures and pipelines with these techniques [38].

REFERENCES

[1] D.Gizopoulos, S.Mukherjee, "Dependable Computer Architecture", Special Section, Guest Editorial, IEEE Trans. on Computers, Jan. 2011.

[2] S.Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation", IEEE Micro, vol.25, no.6, pp. 10-16, Nov.-Dec. 2005.

[3] D.J.Sorin, Fault Tolerant Computer Architecture, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publish., 2009.

[4] N.Aggarwal, P.Ranganathan, N.P.Jouppi, and J.E.Smith. "Configurable isolation: building high availability systems with commodity multi-core processors", ISCA 2007.

[5] C.LaFrieda, E.Ipek, J.F.Martinez, R.Manohar, "Utilizing Dynamically Coupled Cores to Form a Resilient Chip Multiprocessor", DSN 2007.

[6] S.S.Mukherjee, M.Kontz, and S.K.Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives", ISCA 2002.

[7] M.Gomaa, C.Scarbrough, T.N.Vijaykumar, and I.Pomeranz, "Transient-fault recovery for chip multiprocessors", ISCA 2003.

[8] J.C.Smolens, B.T.Gold, B.Falsafi, and J.C.Hoe, "Reunion: Complexity-effective multicore redundancy", MICRO 2006.

[9] N.Oh, P.P.Shirvani, E.J.McCluskey, "Error detection by duplicated instructions in super-scalar processors", IEEE Trans. on Reliability, vol.51, no.1, pp.63-75, Mar 2002.

[10] G.A.Reis, et al., "SWIFT: Software Implemented Fault Tolerance", Intl. Symp. on Code Generation and Optimization (CGO), 2005.

[11] G.Hetherington, et al., "Logic BIST for large industrial designs: real issues and case studies", ITC 1999.

[12] S.Shyam, et al.,,"Ultra Low-Cost Defect Protection for Microprocessor Pipelines", ASPLOS 2006.

[13] M.Psarakis, D.Gizopoulos, E.Sanchez, and M.Sonza Reorda, "Microprocessor Software-Based Self-Testing", IEEE Design & Test of Computers, vol. 27, no. 3, pp. 4-19, May/June 2010.

[14] A.Apostolakis, D.Gizopoulos, M.Psarakis, A.Paschalis, "Software-Based Self-Testing of Symmetric Shared-Memory Multiprocessors", IEEE Trans. on Computers, vol. 58, no. 12, pp. 1682-1694, July 2009.

[15] N.Foutris, et al., "MT-SBST: Self-Test Optimization in Multithreaded Multicore Architectures", ITC 2010.

[16] K.Constantinides, O.Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation", MICRO 2007.

[17] T.M.Austin. "DIVA: A reliable substrate for deep submicron micro-architecture design", MICRO 1999.

[18] C.Weaver and T.Austin, "A Fault Tolerant Approach to Microprocessor Design", DSN 2001.

[19] A.Meixner, M.E.Bauer, and D.J.Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores", MICRO 2007.

[20] A.Meixner and D.J.Sorin, "Error Detection via Online Checking of Cache Coherence with Token Coherence Signatures", HPCA 2007.

[21] R.Fernandez-Pascual, J.M.Garcia, M.E.Acacio, J.Duato, "Low Overhead Fault Tolerant Coherence Protocol forCMP Architectures", HPCA 2007.

[22] K.Chen, S.Malik, and P.Patra, "Runtime validation of memory ordering using constraint graph checking", HPCA 2008.

[23] A.DeOrio, I.Wagner, and V.Bertacco, "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," HPCA2009.

[24] A.Meixner and D.J.Sorin, "Dynamic Verification of Memory Consistency in Cache-Coherent Multithreaded Computer Architectures." IEEE Trans. on Dependable and Secure Computing, vol. 6, no 1, 2009.

[25] P.Racunas, K.Constantinides, S.Manne, and S.S.Mukherjee, "Perturbation-based Fault Screening", HPCA 2007.

[26] N.J.Wang and S.J.Patel, "ReStore: Symptom-Based Soft Error Detection in Microprocessors", IEEE Trans. on Dependable and Secure Computing, 3(3), pp. 188-201, July-Sept 2006.

[27] M.-L.Li, et al., "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design", ASPLOS 2008.

[28] D.J.Sorin et al., "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery", ISCA 2002.

[29] M.Prvulovic, Z.Zhang, and J.Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors", ISCA 2002.

[30] J.Nakano et al, "ReVive I/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers", HPCA 2006.

[31] P.Shivakumar, S.W.Keckler, C.R.Moore, and D.Burger, "Exploiting microarchitectural redundancy for defect tolerance", ICCD 2003.

[32] E.Schuchman and T.N.Vijaykumar, "Rescue: A microarchitecture for testability and defect tolerance", ISCA 2005.

[33] J.Srinivasan, S.V.Adve, P.Bose, and J.A.Rivers, "Exploiting structural duplication for lifetime reliability enhancement", ISCA 2005.

[34] A.Meixner, D.J.Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," DSN 2008.

[35] J.Collet, P.Zajac, M.Psarakis, and D.Gizopoulos, "Chip Self-Organization and Fault-Tolerance in Massively Defective Multicore Arrays", IEEE Trans. on Dependable and Secure Computing, 2010.

[36] S.Gupta, et al., "The StageNet fabric for constructing resilient multicore systems," MICRO 2008.

[37] B.F.Romanescu and D.J.Sorin, "Core cannibalization architecture: Improving lifetime chip performance for multicore processors in the presence of hard faults", PACT 2008.

[38] M.Powell, A.Biswas, S.Gupta, S.Mukherjee, "Architectural Core Salvaging in a Multi-Core Processor for Hard Error Tolerance", ISCA 2009.

[39] S.Hari et al, "Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems", MICRO 2009.

[40] S.Sahoo et al, "Using Likely Program Invariants to Detect Hardware Errors", DSN 2008.

[41] M.Li et al., "Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults", DSN 2008.

[42] F.A.Bower, P.G.Shealy, S. Ozev, and D. J. Sorin, "Tolerating Hard Faults in Microprocessor Array Structures", DSN 2004.