mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems [*]

Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, Sarita V. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
swat@cs.uiuc.edu

ABSTRACT

Continued technology scaling is resulting in systems with billions of devices. Unfortunately, these devices are prone to failures from various sources, resulting in even commodity systems being affected by the growing reliability threat. Thus, traditional solutions involving high redundancy or piecemeal solutions targeting specific failure modes will no longer be viable owing to their high overheads. Recent reliability solutions have explored using low-cost monitors that watch for anomalous software behavior as a symptom of hardware faults. We previously proposed the SWAT system that uses such low-cost detectors to detect hardware faults, and a higher cost mechanism for diagnosis. However, all of the prior work in this context, including SWAT, assumes single-threaded applications and has not been demonstrated for multithreaded applications running on multicore systems.

This paper presents mSWAT, the first work to apply symptom based detection and diagnosis for faults in multicore architectures running multithreaded software. For detection, we extend the symptom-based detectors in SWAT and show that they result in a very low Silent Data Corruption (SDC) rate for both permanent and transient hardware faults. For diagnosis, the multicore environment poses significant new challenges. First, deterministic replay required for SWAT's single-threaded diagnosis incurs higher overheads for multithreaded workloads. Second, the fault may propagate to fault-free cores resulting in symptoms from fault-free cores and no available known-good core, breaking fundamental assumptions of SWAT's diagnosis algorithm. We propose a novel permanent fault diagnosis algorithm for multithreaded applications running on multicore systems that uses a lightweight isolated deterministic replay to diagnose the faulty core with no prior knowledge of a known good core. Our results show that this technique successfully diagnoses over 95% of the detected permanent faults while incurring low hardware overheads. mSWAT thus offers an affordable solution to protect future multicore systems from hardware faults.

Categories and Subject Descriptors

B.8.1 [**Reliability, Testing and Fault-Tolerance**]

General Terms

Reliability, Experimentation, Design

Keywords

Error detection, Multicore processors, Architecture, Fault injection

1. INTRODUCTION

Driven by Moore's Law, continuous device scaling has provided smaller devices that make increasing system integration feasible. These smaller devices are, however, susceptible to failures due to various phenomena such as high energy particle strikes, aging or wear-out, infant mortality, and so on [2]. Since this reliability threat is projected to affect the broad computing market, traditional solutions involving excessive redundancy are too expensive in area, power, and performance [1, 26].

Answering this need for low-cost solutions, previous work has proposed the use of high-level symptom detectors that identify only those hardware faults that propagate to higher levels of the system [4, 5, 10, 12, 14, 19, 22, 23, 28, 30]. Compared to traditional redundancy based schemes, such detection solutions incur lower overheads, but assume checkpoint/rollback support for recovery. Much of the prior work in this context has focused on transient faults [4, 5, 17, 19, 22, 28, 30] where the combination of detection and recovery forms a complete reliability solution. In contrast, permanent faults, that are becoming increasingly important [10, 14, 23], further require a diagnosis mechanism to isolate the faulty core or microarchitectural component for repair or reconfiguration.

In particular, the SWAT (SoftWare Anomaly Treatment) project [9, 10, 23] has undertaken a comprehensive exploration of the above approach for detection, diagnosis, and recovery for both permanent and transient hardware faults. SWAT has shown that a small set of simple and high-level detectors can provide very high detection coverage at negligible cost [10, 23]. Further, SWAT uses a synergistic diagnosis algorithm, called Trace Based Fault Diagnosis (TBFD), that leverages the recovery mechanism to distinguish between software bugs, and permanent and transient hardware faults, and to isolate the faulty microarchitectural component in the case of permanent faults [9]. SWAT relies on existing checkpoint/rollback mechanisms for recovery [21, 25], and employs existing microachitecture-level techniques for repair/reconfiguration in the case of permanent faults [3, 6, 27]. The key SWAT philosophy is that detection must incur minimal overhead since it is always

[*]This work is supported in part by the Gigascale Systems Research Center (funded under FCRP, an SRC program), the National Science Foundation under Grants NSF CCF 0541383, CNS 0720743, and CCF 0811693, an OpenSPARC Center of Excellence at Illinois supported by Sun Microsystems. Pradeep Ramachandran is supported by an Intel PhD fellowship and an IBM PhD scholarship.

MICRO '09, December 12–16, 2009, New York, NY, USA.

on while diagnosis can afford higher overhead (but within the acceptable mean time to repair) because it is rarely invoked.

Although SWAT has been shown to be largely successful, a current limitation of SWAT, and all the above cited work, is the assumption of single-threaded workloads running on a single core. Multithreaded software running on multicore systems is, however, becoming increasingly important. Thus, for a system like SWAT to be widely deployable, it must include techniques to detect and diagnose faults in multicore systems.

SWAT's symptom-detection can potentially be used to detect faults in the new environment, but their efficacy needs to be evaluated. The original SWAT diagnosis algorithm, TBFD, however cannot be directly applied for two reasons. First, TBFD requires deterministic replays of the software execution for diagnosis, which is non-trivial for multithreaded software. While recent methods for deterministic replay of multithreaded software may be leveraged [7, 15, 18, 31] the accompanying overheads are too high. Second, because of data sharing across threads, a fault may escape from a faulty core to a fault-free core (a phenomenon we call *cross-core fault propagation*), resulting in breaking two assumptions of TBFD: (1) the symptom causing core can no longer be assumed to be faulty, and (2) once a fault is detected, no core in the system can be assumed to be fault-free.

This paper presents mSWAT – the SWAT approach for multithreaded workloads running on multicore systems in the presence of faults. We investigate the efficacy of the low-cost detectors of SWAT and develop a novel diagnosis algorithm for multithreaded workloads on multicore. We leverage existing work on multicore recovery to recover these faults [21, 25]. Our contributions are:

1. **Detection:** We show that with a small extension, the existing SWAT detectors are very effective on multicore processors. The augmented detectors result in a low SDC rate of 0.2% for permanent faults and 0.5% for transient faults. A large fraction of the detected faults are detected within 10M instructions, and can be recovered using previously proposed methods. Further, 4.5% of the detected permanent faults cause symptoms from fault-free cores, confirming the need for the diagnosis algorithm to handle such situations.

2. **Diagnosis:** We propose a novel algorithm to identify the faulty core in the case of a permanent fault with no prior knowledge of a known good core, even in the presence of cross-core fault propagation. Using a lightweight technique to deterministically replay each thread of the multithreaded workload in *isolation*, the algorithm synthesizes an inexpensive selective Triple Modular Redundant (TMR) replay to achieve a scalable solution that incurs minimal hardware overheads. Our algorithm successfully diagnoses over 95% of the $7,449$ detected permanent faults. In particular, all the faults that resulted in symptoms in fault-free cores (due to cross-core fault propagation) are successfully diagnosed. Additionally, 96% of the diagnosed faults require $< 200KB$ hardware buffers for diagnosis. This can be implemented in lower level caches of modern processors, incurring low hardware overheads.

To the best of our knowledge, this is the first work that provides a low-cost detection and diagnosis of hardware faults in multithreaded workloads running on multicore systems, without relying on expensive, always-on redundancy. This work uses redundancy only for diagnosis, which is a relatively rare case. Fault-free operation, which remains the common case, continues to see near-zero overhead. Although our work is presented here in the context of SWAT, the diagnosis algorithm can be used in combination with other high-level detection mechanisms that may allow a fault to escape the faulty core.

2. RELATED WORK

Reliable system design has been a prominent area of research for many decades. There has been much recent work on using high-level symptom based detection solutions that provide substantial reliability benefits for little cost [4, 5, 10, 14, 17, 19, 22, 23, 28, 30]. We focus on SWAT as exemplifying this line of work and discuss it in more detail below. We also discuss related work in checkpointing and deterministic replay techniques. Other related work includes diagnosis for the DIVA architecture [3] and low-level online testing based diagnosis [24] – both techniques do not consider faults escaping the faulty core in multicore environments and both incur overheads during fault-free operation.

2.1 SWAT: SoftWare Anomaly Treatment

Two high-level observations drive the SWAT work [10]. First, any hardware reliability solution should handle only those faults that affect software execution. Second, despite the growing reliability threat, fault-free operation remains the common case and must be optimized. These observations motivate fault detection by watching for anomalous software behavior using zero to low-cost hardware and software monitors. With this approach, the fault detection mechanism is largely oblivious to the underlying hardware fault mechanism, treating hardware faults analogous to software bugs and potentially leveraging solutions for software reliability to amortize overhead. Given that fault detection occurs at a high level, diagnosis is more complex. However, since diagnosis is invoked only in the relatively rare event of a fault, a higher overhead is acceptable (but within the constraints of mean time to repair). We describe the detection and diagnosis mechanisms of SWAT below.

Fault Detection: The initial SWAT work proposed hardware fault detection that employ very low-cost monitors to detect anomalous software behavior with very little hardware and no software support [10]. mSWAT uses these hardware-only detectors, with one new addition, as described in Section 3. Subsequently, the iSWAT framework proposed using mined likely program invariants from the application to augment these hardware-only detectors [23]. Although these invariants further reduce the SDC rate, we do not explore these detectors here as they required changes to the application binary.

Fault Diagnosis: After a symptom is detected, control is transferred to the firmware that initiates diagnosis. Since the detectors detect faults at a high level, the diagnosis mechanism should distinguish between software bugs, transient and permanent hardware faults, and false positives (from iSWAT and a few heuristic detectors), and in the case of a permanent fault, identify the faulty microarchitectural component for repair. TBFD, the diagnosis algorithm of SWAT, achieves this with the help of another fault-free core and the ability to deterministically replay an execution of the single-threaded application. The key insight here is that the execution that generated the symptom can be used as a *test trace* to repeatedly activate any faults present in order to incrementally perform diagnosis. SWAT leverages existing recovery techniques to repeatedly rollback, replay and compare the test trace from the faulty core to that from a known good core for diagnosis.

TBFD first uses three simple steps to distinguish between software bugs, transient faults, and permanent faults: (1) It first replays the symptom-generating execution from the last checkpoint on the same (faulty) core. If the symptom does not recur, a transient fault or a non-deterministic software bug is diagnosed and the execution continues. (2) If the symptom recurs, then the fault-free core executes the thread from the same checkpoint. If the symptom also occurs in the fault-free core, then the fault is diagnosed as a software bug. (3) If no symptom occurs in the fault-free core, the original core is diagnosed with a permanent fault.

TBFD further diagnoses a permanent fault down to a microarchitectural granularity to facilitate fine-grained repair and reconfiguration. TBFD inexpensively synthesizes Dual-Modular Redundancy (DMR) between the known faulty and good cores, comparing the execution traces on the two cores. A mismatch between the executions helps determine the faulty microarchitectural component. TBFD successfully diagnoses 98% of the faults detected in single-threaded workloads [9], making it a highly effective diagnosis tool.

2.2 Checkpointing and Deterministic Replay

Since mSWAT diagnosis is based on repeated rollback/replay, we leverage existing work for rollback recovery on multicore systems [21, 25]. These schemes use a combination of checkpointing and logging techniques to establish periodic checkpoints of the processor and memory state. For error recovery, the rollback procedure restores the pristine processor and memory state from the checkpointed state and logged states.

The diagnosis algorithm in mSWAT requires the ability to deterministically replay a core's execution from its previous checkpoint, in isolation from the other cores. While there has been much recent work on determinisic replay of multithreaded workloads [7, 15, 18, 31], our work in this context is closest to BugNet [18] but with key differences, as described below.

BugNet [18] (and other schemes [7, 15, 31]) were designed for software production runs, where the continuously collected logs are transferred back to the developer on a crash. The program can then be deterministically replayed to determine the root cause of the crash. To efficiently replay all application threads, BugNet logs only the first loads to a given memory address and cross-thread communications through memory.

We leverage the idea of logging the load values from BugNet to enable isolated deterministic replay, but with the following key differences. First, our scheme logs all loads because logging only the first load values would not suffice as the hardware fault may corrupt the value returned by subsequent loads. This also gives us the ability to replay each thread in isolation as loads do not access memory during replay. Second, these logged loads circumvent recording memory orderings, reducing the incurred area overhead. We also present an iterative techinque to further reduce the sizes of these logs (Section 4.2.5). Finally, since our logging is turned on only in the rare event of a fault, it does not affect fault-free operations and can tolerate higher performance overheads, which we leverage for lower area overheads.

3. MSWAT FAULT DETECTION

mSWAT uses the following detectors to detect hardware faults in multicore systems. The first three detectors were previously proposed for SWAT [10]. We extend this list with a new detector to identify kernel panics.

- **Fatal-Traps**: These traps are not thrown in normal execution and indicate anomalous software behavior (e.g., Division by Zero). These are thus used as zero-cost detectors to detect hardware faults.
- **Hangs**: A simple hardware hang detector, that monitors the frequency of branch instructions in the application and OS, is used to identify hangs.
- **High-OS**: Typical invocations of the OS, except for system calls and interrupts, complete in few 100s of instructions. Thus, abnormally high number of contiguous OS instructions represents an anomaly. Existing performance counters can trivially identify such scenarios.
- **Panic**: When the kernel detects an unrecoverable error during execution, it self-terminates by calling a known centralized panic reporting routine to minimize potential damage to the user data and to facilitate debugging. This detector, thus, monitors the PC of the retiring instructions to identify kernel panics, and can be implemented with simple support from the OS and the hardware.

Once a fault causes a symptom, it invokes the mSWAT firmware that initiates diagnosis to identify the faulty component.

4. MSWAT FAULT DIAGNOSIS

In this work, we only consider faults in the core and assumes a single core fault model; i.e., at most one core is faulty. The mSWAT multicore diagnosis procedure must achieve the following:

1. Determine whether the symptom was caused by a software bug, a transient hardware fault, or a permanent hardware fault. Although SWAT also does this, the new multithreaded environment poses new challenges to mSWAT.
2. For permanent faults, determine which core is faulty even with cross-core fault propagation. In contrast, SWAT simply assumes that the symptom-causing core is faulty.
3. Depending on the granularity of the field reconfigurable unit, isolate the faulty microarchitecture-level unit in the core for repair. To achieve this, mSWAT uses TBFD [9] after identifying the core that contains a permanent fault.

Recall that the key insight behind TBFD is to use the execution that generated the symptom as a *test trace*; repeated replays and comparisons of this test trace on good and faulty cores result in a diagnosis. A multithreaded environment creates two main challenges to exploit this insight for mSWAT. First, deterministic replay of multithreaded workloads incurs high hardware overheads, that may be unacceptable. Second, cross-core fault propagation breaks two fundamental assumptions of TBFD – the symptom causing core can no longer assumed to be faulty, and once a symptom is detected, no core in the system can be assumed to be fault-free.

To address the first issue, existing proposals for deterministic replay of multithreaded software [7, 15, 18, 31] can be leveraged, but their hardware cost needs to be brought down. Deriving a cost-effective method to address the "no known good core" issue, on the other hand, is a significant challenge.

A naive extension of the current diagnosis scheme in SWAT for a multithreaded execution running on N cores would use this N-core execution as the test trace. For identifying the faulty core, it would rollback this execution to the last checkpoint and perform a full system deterministic replay on another set of N (known) good cores. A comparison of corresponding cores for these two executions would identify the faulty core. This algorithm must assume that N known good (spare) cores and a facility for full deterministic replay are available, which is clearly too expensive for most systems.

An optimization is to use only one spare (known-good) core for a total of $N + 1$ cores. This allows N replays from the N-core checkpoint, with each replay replacing one of the original N cores with the known-good spare core. The execution where no symptom occurs identifies the replaced core as the faulty one. This solution also has several drawbacks: (i) it is not scalable because it could require up to N replays, (ii) it requires one known good spare core just for this purpose, making it the single point of failure for the entire system, and (iii) it also requires support for full deterministic replay.

We propose an algorithm with the following desirable properties: (1) *No spare cores are required* – This eliminates a potential single point of system failure. (2) *Low hardware overhead* – the algorithm uses a lightweight deterministic replay mechanism that does not require capturing memory ordering among different threads. (3) *Scalability* – the algorithm diagnoses a faulty core with a maximum of 3 replays (plus one replay to screen out transients) for any system with $N \geq 4$ cores.

4.1 mSWAT Fault Diagnosis Overview

mSWAT addresses the two challenges above as follows. For deterministic replay of the multithreaded execution, we note that the diagnosis has no need to recreate the memory ordering among different cores but only cares about the per-core execution that activates the fault (test trace). Hence, during diagnosis, mSWAT collects enough information in the execution of each core to replay the core's trace in isolation. For this purpose, we leverage BugNet's idea of using load values to replay the execution on each core. BugNet records only the first loads to a given memory address, along with memory ordering of different threads, and relies on the execution to generate subsequent values at that address. However, since the fault may corrupt the values subsequently generated at that memory location, we log the values accessed by *all* loads in the system. With this approach, our diagnosis scheme is able to replay each trace independently on a different core. A divergence between the original trace and the replay then provides two candidates for the faulty core. Although we start without the knowledge of a known good core, this divergence indicates that the other cores are fault-free (since we assume at most one core is faulty). Hence, another replay on one of the known good cores can pinpoint the true faulty core. Overall, we can view mSWAT's algorithm as an inexpensive synthesis of Triple Modular Redundancy (TMR) since it compares up to three different executions to identify a fault.

In mSWAT, fault diagnosis is done in four phases, as illustrated in Figure 1 – screening, trace generation, first replay, and (possibly) second replay. We give a high level overview of each of these phases below and subsequently discuss various implementation choices for each phase. As with SWAT, mSWAT leverages a recovery method like SafetyNet or Revive, assuming a mechanism that can safely roll back to a previous pristine checkpoint and restart execution.

Screening: The first phase screens for transients and non-deterministic software bugs by replaying the execution on all cores from the previous pristine checkpoint.[1] This simple rollback/replay may not be deterministic because of different thread interleavings in multithreaded executions. If this replay does not result in any symptom, we diagnose the fault as a transient or a non-deterministic software bug and simply let the execution continue. If a symptom is seen in screening, we suspect the fault to be either a permanent hardware fault or a deterministic software bug.

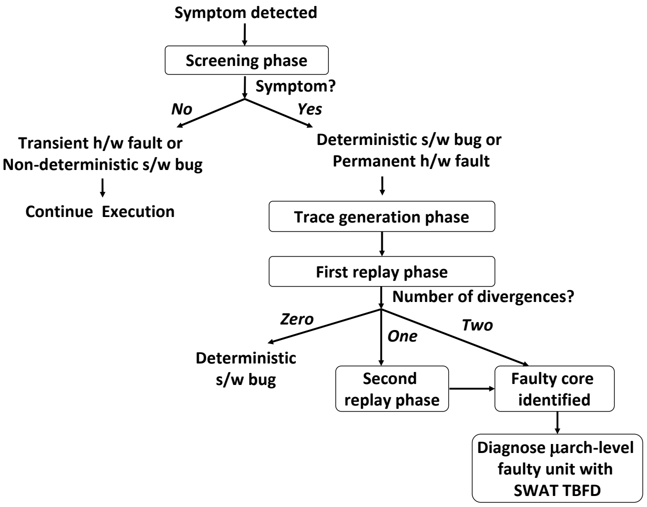


Figure 1: The mSWAT diagnosis algorithm. mSWAT isolates the source of the fault by replaying the fault activating trace and comparing the re-executions.

Trace generation: Using the recovery mechanism, the previous checkpoint is again restored and the execution is replayed. In this phase, each core stores a trace of its execution that contains enough information to (i) enable future deterministic replay from the same checkpoint (load values in our case) and (ii) compare subsequent replays of this trace and identify divergences caused by activations of an underlying permanent fault. We refer to the information in the trace needed for the first part as the *loadLog* and the second part as the *compareLog*. An implementation may choose to either merge or separate these logs, as discussed later.

First replay: This phase aims to replay the execution of each thread and to diagnose the faulty core by identifying divergences. Each core is assigned a buddy core from which it gets a checkpoint and a trace for replay. Once a core finishes generating its own trace, it cedes control to firmware to wait for its buddy core to finish trace generation (indicated through a flag synchronization mechanism). The core then loads the checkpoint of the buddy and deterministically replays the execution using the loadLog of the buddy. The checkpoint in this case is only the core state checkpoint (registers); memory state does not need to be recovered since all execution-driven data comes from the loadLog in the trace. For the same reason, in this phase, stores do not need to write to memory and are effectively no-ops. The compareLog generated in this replay is compared with that received from the buddy to identify divergences during replay.

There are three possible outcomes of the comparison. (1) If there is no divergence (for all cores), a software bug is assumed and control is delivered to the relevant appropriate software layer. (2) If two replays (two pairs of cores) show divergence in this phase, then the core that is in both the diverging pairs is faulty. This can happen when the fault is activated in both the trace generation and the first replay phases, resulting in divergences from two traces. After iden-

[1] This pristine checkpoint is at least next to last because the last one may be corrupted by the fault.

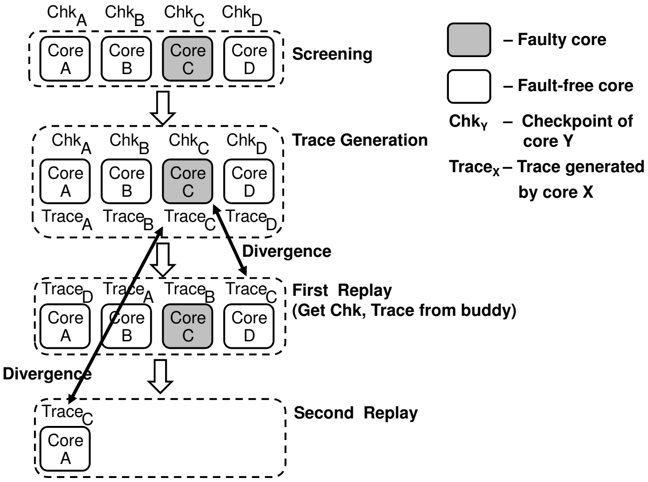

Chk$_A$ Chk$_B$ Chk$_C$ Chk$_D$
Core A
Core B
Core C
Core D
Screening
Trace Generation
Trace$_A$ Trace$_B$ Trace$_C$ Trace$_D$
Divergence
Trace$_D$ Trace$_A$ Trace$_B$ Trace$_C$
First Replay
(Get Chk, Trace from buddy)
Divergence
Trace$_C$
Second Replay
– Faulty core
– Fault-free core
Chk$_Y$ – Checkpoint of core Y
Trace$_X$ – Trace generated by core X

Figure 2: Example of the mSWAT diagnosis mechanism. Isolated deterministic replay allows synthesizing selective TMR to diagnose the faulty core.

tifying a faulty core, the diagnosis algorithm terminates. (3) If only one divergence is seen, then two suspected faulty cores are found, the remaining cores are declared to be fault-free and the algorithm proceeds to the second replay phase.

Second replay: The trace that saw the divergence is replayed on a core that is now known to be fault-free. This replay is similar to the first replay phase and is compared against the compareLog generated in the trace generation phase to identify divergence. If a divergence occurs, the core that generated the trace is faulty. If no divergence occurs, the core that replayed this trace in the first replay phase is faulty.

Figure 2 illustrates the above diagnosis procedure with an example to diagnose the fault in $Core_C$. Once a symptom is seen in the screening phase, trace generation is triggered. During this phase, $Core_{A-D}$ generates $Trace_{A-D}$, respectively, comprising a loadLog and a compareLog of the corresponding execution. Subsequently, in the first replay phase, each core uses the (core-only) checkpoint and trace of the core to its left to replay. Now $Core_D$ sees a divergence since it replays the trace from the faulty $Core_C$. Since we cannot be sure which core is faulty, this event identifies two potentially faulty suspects, $Core_C$ and $Core_D$, and invokes the second replay phase. Here, $Core_A$, a known fault-free core, replays from $Checkpoint_C$ using $Trace_C$. It finds a divergence during replay, confirming that $Core_C$ is faulty.

4.2 Implementation Details

This section describes each of the above described phases in more detail, highlighting several design choices needed to perform the diagnosis in a reliable way and with acceptable overheads.

4.2.1 Screening

The screening phase is similar to a typical rollback/replay recovery. To determine whether a symptom re-occurred during screening, the replay controller in the recovery mechanism is enhanced with a hardware counter that tracks the number of cycles since the last symptom. If a symptom occurs on any core within a pre-defined threshold, the controller determines this to be a recurring symptom and sends an interrupt to all cores to invoke the trace generation phase. (If no symptom is seen, the cores simply continue.)

4.2.2 Trace Generation

The trace generation phase also starts like a recovery – a rollback and a replay. In addition, each core also generates a loadLog and a compareLog for isolated deterministic replay and for comparison to identify divergence. We discuss design choices in this phase below.

1. **Information for Replay:** As discussed in Section 4.1, we record the values of all retiring loads to drive deterministic replay. To reduce the log size, we can use BugNet's dictionary structure to encode load values provided that the added hardware cost to generate such encoding is acceptable. We do not explore this optimization here.

   To avoid asynchronous events that may interfere with deterministic replay, we disable interrupts during the trace generation and replay phases. An exception is the interrupts generated by the firmware controlling the diagnosis algorithm. Unlike BugNet, our scheme traces and replays system calls as well because we log all loads, including privileged loads. Our scheme, however, does not consider self-modifying code since we assume that the instruction at a given PC is fixed across different replays.

2. **Information for Trace Comparison:** A naive implementation to compare the traces generated across different replays is to record the destination values of *all* retiring instructions. We reduce the resulting overhead by recording and comparing only data and address values of stores and the branch targets of retiring control instructions in the compareLog and the address of loads in the loadLog.[2]

   Logging every store and branch may also result in large compareLogs. One possible optimization is to instead generate signatures that represent the entire execution (e.g., CRC-16). This approach requires hardware support for signature computation, impacting the overheads and diagnosis latency as the divergence status is not known until the entire trace is replayed. (By comparing instructions during replay, a divergence is known right away.) We reduce the hardware overhead for recording load addresses by recording only a parity bit (generated by a parity module) instead of the entire 64-bit load address for trace comparison.

   Although we lose some diagnosis opportunity through these optimizations, our results show that the benefits outweigh the losses.

3. **The Trace Buffer and Other Details:** The simplest implementation of the trace buffer is a small FIFO buffer that is memory backed and contains a merged loadLog and compareLog. As the core retires loads, stores, and branches, the relevant information is recorded in the trace buffer. The buffer is periodically flushed to memory, similar to analogous buffers in BugNet and TBFD. Since diagnosis can accommodate some performance slack, this hardware buffer can be as small as a few entries; the actual size is determined by the acceptable hardware cost and sensitivity to diagnosis latency. In practice, we expect most of the buffer will reside in cache to minimize off-chip memory bandwidth requirements (Section 6 shows the actual trace sizes for our design choice).

   We note here that having the trace buffer be memory backed makes the generated traces vulnerable to corruptions by the

[2] While this load address is technically a part of the compareLog, we record it in the loadLog to simplify our implementation.

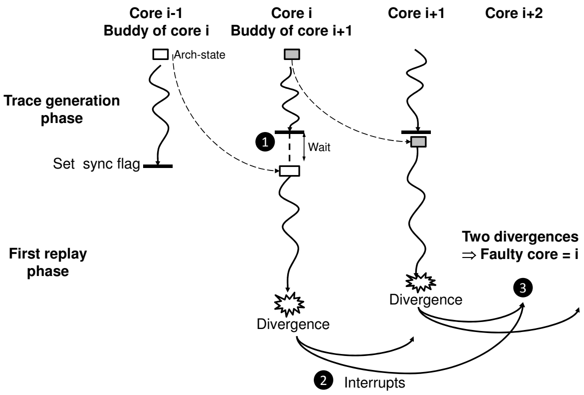


Figure 3: An example of first replay phase where two divergences are observed.

faulty core. This problem occurs with the basic recovery schemes as well [21, 25]. One solution is to provide hardware support to ensure that no core writes to the trace region of another core through simple base and bounds-style checks.

The trace generation phase starts from a checkpoint and continues until the number of instructions exceeds a pre-defined threshold (or until a symptom is detected). Since the log size grows with the threshold, we later describe an iterative method that alternates between trace generation and replay to reduce the size of the log.

4.2.3 First Replay

Figure 3 illustrates the first replay phase. At the end of a core's trace generation, the firmware seizes control, sets a flag, and waits for the buddy core to set its flag. A core's buddy is the core from whom it gets the checkpoint/trace to replay – this buddy is predefined, say core (i+1) mod N replays the trace for core i, where N is the number of cores. (In the subsequent discussion, we drop the *mod N* for brevity.)

Thus, core i waits for core i-1 to set its flag to indicate it has finished its trace generation (❶ in Figure 3). Once the flag is set, the firmware begins replay on core i. Although the faulty core may try to subvert this procedure by not setting its flag, a time-out mechanism can detect such a subversion and terminate the procedure with a correct diagnosis.

The simplest implementation to replay and compare the replayed execution with that of the buddy is to have the firmware *emulate* the execution using the register checkpoint and perform the comparison all in software.[3] During emulation, the loads return values from the loadLog. Branches, stores, and load addresses are compared against the information fetched from the buddy's trace. If there is no divergence between the replayed execution and the buddy's replay, the core sets a flag and waits for the other cores as described below.

If a divergence occurs, it indicates that either this core, say i, or its buddy, i-1, is faulty. More importantly, this implies that the other cores are fault-free (since we assume a single core fault model).

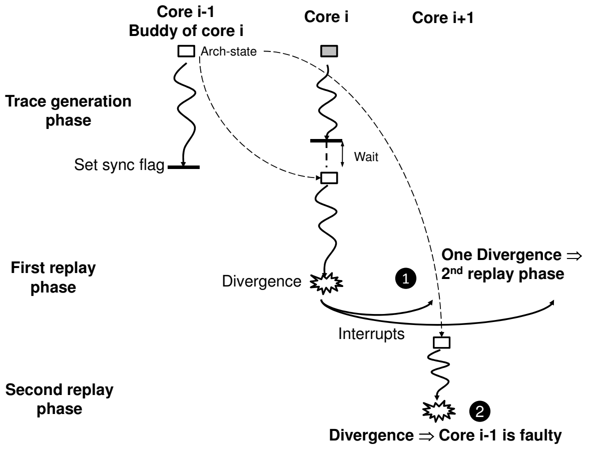


Figure 4: An example of second replay phase where a divergence is observed.

Since core i itself could be faulty and can no longer be relied on, it sends an interrupt to two cores, i+1 and i+2 (which are now known to be fault-free), to take over the process (❷ in Figure 3).

We need two cores to take over now to prevent the faulty core from subverting the process. For example, if core i were faulty, it may not send the above interrupt to core i+1 and it may appear that we will lose diagnosability. This is not a problem because core i+1 itself will see a divergence (since it is comparing a faulty trace), and will inform core i+2, which is a good core. The more problematic case is when the faulty core sends an interrupt to a different core (core i+k) masquerading as core i+k-1, and indicating that good cores i+k-1 or its buddy are faulty. To overcome this, we assume that the core issuing this interrupt cannot fake its identity.

Thus, at the end of the first replay, if a divergence was reported to a good core, then at least two cores that are known to be good have received an interrupt indicating that a divergence has been found. If one of these good cores received notice of two divergences, then it could deduce that the common core involved was faulty (❸ in Figure 3). It informs another fault-free core – together they deconfigure the faulty core, and set a flag each to indicate that the bad core is found. (Two cores are needed for this procedure because the faulty core could otherwise deconfigure a good core and set a bad flag.) If only one divergence is seen, then these two good cores enter the second replay phase to isolate the faulty core.

At the end of this phase, if no divergence is found, flags corresponding to all cores are set, and a deterministic software bug is diagnosed. Since the faulty core may not set this flag, a timeout is triggered when only N-1 flags are set. Subsequently, the faulty core is diagnosed and two pre-specified cores (faulty+1 and +2) can deconfigure that core.

4.2.4 Second Replay

At this point, two suspected faulty cores (i-1 and i) are identified and two cores (i+1 and i+2), known to be fault-free, are notified of this event (❶ in Figure 4). Core i+1 now replays and compares with the trace generated by core i-1 for a divergence. If a divergence is found (❷ in Figure 4), core i-1 is diagnosed as faulty as it diverges from both core i and i+1. If no divergence is found, core i is diagnosed as faulty since core i-1 and i diverge but fault-free core i+1 does not diverge from the trace of core i-1. Subsequently, both i+1 and i+2 set their flags appropriately and deconfigure the faulty core. As an alternative, core i+1 or i+2 can also invoke TBFD on the faulty core to achieve a finer granularity of diagnosis and repair

[3] An alternative, faster implementation is to replay natively with hardware support and stream in the loadLog and compareLog to on-chip buffers. This requires changes in the processor pipeline since loads now need to read from the loadLog buffer in the issue stage. The benefit is that it does not incur the penalties of emulation; however, the disadvantage is hardware cost. Since our first priority is to minimize hardware overhead, we do not report results from this implementation here.

instead of disabling the entire core. Once all the flags have been set, diagnosis terminates and the recovery mechanism is invoked.

4.2.5 Iterative Diagnosis Approach

While the described method can effectively identify a faulty core, large traces may incur unacceptably high area and performance overheads. To limit these overheads, we propose an *iterative approach* where the trace generation and the first replay phases are executed repeatedly on short traces until a divergence is observed or a predefined number of instructions is executed. Once a divergence is identified, the second replay phase is invoked to identify the faulty core as before.

At the end of each (short) trace generation phase, the state of the processor is checkpointed and the first replay phase is initiated to identify any divergence. If no divergence is found, the checkpointed processor states are restored and the next iteration of trace generation is performed (followed by a subsequent replay and so on). Since stores do not write to memory in the replay phases, memory checkpointing and rollback is not necessary between iterations. (However, memory checkpointing remains on during trace generation.)

Shorter iteration lengths make this iterative diagnosis method more effective by reducing the size of the trace. However, executing fewer instructions in each iteration may result in short traces that may not utilize the microarchitecture fully, leading to fewer fault activations, and hence, fewer divergences and loss of diagnosability. This is an inherent design trade-off for the iterative approach.

4.2.6 Other Issues

As with every reliability solution, we need to ensure that the faulty core and the firmware running on it do not subvert our algorithm. Further, we need to identify the hardware that we rely on to be fault-free. Although we do not provide a formal proof here, our algorithm description indicates how we guard against subversion by the faulty core in the critical places as well as the small amount of hardware that we need to work reliably.

5. EXPERIMENTAL METHODOLOGY

Ideally, we would perform hardware fault injections on a full implementation of mSWAT. However, since modern processors do not allow sufficient controllability and observability and since we do not yet have the required models to perform FPGA-based fault injections [20], we turn to simulations for our experiments, much like SWAT [10]. While we previously showed that injecting faults at the microarchitecture (vs. gate) level could result in some inaccuracies [8], we choose microarchitecture-level fault injection since we do not have gate level models of all modules of interest. Since a large part of our focus is on diagnosing permanent faults, we primarily focus on such faults here. Following previous work, we use microarchitecture level single bit stuck-at-0 and stuck-at-1 fault models. For completeness, we also perform detection experiments for transients (modeled as single bit flips) and summarize those results here.

5.1 Simulation Environment

We simulate a modern multicore processor with the SPARC V9 ISA, having four out-of-order superscalar cores (Table 1). We use the Virtutech Simics full system simulator [29] coupled with the GEMS [13] timing models for processor and memory. We run a real operating system (OpenSolaris) within this simulated environment and study the behavior of six multithreaded workloads from three benchmark suites. Table 2 lists the workloads we use, along with a description of the input sets.

Base Processor Parameters	
Frequency	2.0GHz
Number of cores	4
Per-core parameters	
Fetch/decode/ execute/retire	4 per cycle
Functional units	2 Int add/mul, 1 Int div, 2 Load, 2 Store, 1 Branch, 2 FP add, 1 FP mul, 1 FP div/sqrt
Integer FU latencies	1 add, 4 mul, 24 div
FP FU latencies	4 default, 7 mul, 12 div
Reorder buffer size	128
Register file size	256 integer, 256 FP
Load-store queue	64 entries
Memory Hierarchy Parameters	
Data L1 (private)	16KB
Instruction L1 (private)	16KB
L1 hit latency	1 cycle
L2 (Unified)	4MB
L2 hit/miss latency	6/80 cycles

Table 1: Parameters of the simulated processor.

Suite	**Workload**	**Size of input**
ALPBench	RayTrace	A teapot scene (2560×2560 pixels)
	FaceRec	173 images (130×150 pixels)
	MpegEnc	32 HD frames (1920x1080)
	MpegDec	128 HD frames (1920x1080)
SPLASH-2	LU	1600x1600 matrix
PARSEC	BodyTrack	4 cameras, 4 frames, 4000 particles, 5 annealing layers

Table 2: Workloads and inputs used in fault injections.

When a multithreaded application is running on all 4 simulated cores in the system, we inject hardware faults (one per experiment) in a random bit of the 7 microarchitecture units listed in Table 3. For each application, we first pick 5 base injection points (or *phases*) in the execution that are sufficiently spaced apart to capture application behavior across different periods of execution. In each phase, for each faulty structure, we pick 5 spatially and temporally random injection points (e.g., 5 different physical registers, etc.) to inject permanent (stuck-at-0 and stuck-at-1) and transient faults. This give us a total of 8,400 permanent faults (6 applications $\times$ 5 phases $\times$ 4 cores $\times$ 7 structures $\times$ 5 random points $\times$ 2 fault models), and 4,200 transient faults (same as above, except one fault model – single bit flip).

5.2 Fault Detection

We detect the injected faults using the SWAT and the new mSWAT symptoms. Based on profiling fault-free executions of our workloads, we use a threshold of 50K contiguous instructions to trigger the *High-OS* detector and a threshold of 1% of retiring instructions as branches for the hang detector, avoiding false positives.

μarch structure	**Fault location**
Instruction decoder	Input latch
Integer ALU	Output latch
Register data bus	Bus on register file write port
Physical int reg file	A physical register
Reorder buffer (ROB)	Entry's src/dst reg id
Reg alias table (RAT)	Logical→ physical map of logical register
Address gen unit (AGEN)	Virtual address output

Table 3: Microarchitectural structures injected with faults.

After a fault is injected, we simulate the system (with application and OS running) until each core has retired at least 10 million instructions, a duration we deem recoverable by hardware checkpointing [21, 25]. If the fault does not corrupt the architectural state (registers and memory) in this interval, it is architecturally masked. Faults that are not architecturally masked, and hence *activated*, are simulated in detail until either they trigger one of the detectors or until all cores retire more than 10 million instructions from activation, whichever comes first.

Unmasked faults not detected in this period are simulated in functional mode (using only Simics) until the application completes or a symptom is detected. Since the fault is not active in the functional simulation, the injected permanent fault appears as an intermittent fault that lasts for 10M instructions. Faults detected in this interval are classified as DUE (detected unrecoverable errors) as they are detected at latencies that current hardware checkpointing schemes may not support.

The application outputs of the remaining undetected cases are compared with the fault-free outputs. If the outputs match exactly, the fault is classified to be masked by the application. Other outputs are categorized as Silent Data Corruptions (SDCs). In spite of the differences from their fault-free counterparts, these outputs may still be acceptable, especially by workloads that output multimedia content. Following previous work [11], we classify an output with PSNR of $> 50dB$ as acceptable for MpegEnc and MpegDec. For the other benchmarks, we do not tolerate any differences in application output. Such acceptable outputs are categorized as SDC-acceptable and the rest as SDC-unacceptable.

We use two metrics to evaluate the efficacy of the symptom detectors – SDC rate and latency. *SDC rate* is the fraction of the injected faults that results in unacceptable outputs (SDC-unacceptable). *Detection latency* computation is a bit more involved as the fault may be detected in a fault-free core due to cross-core fault propagation. If the fault is detected in the faulty core, it is the latency (in instructions) between the the architectural state corruption of this core and symptom detection. If the detection is in a fault-free core, we identify the instruction count on the fault-free core at which the architectural state of the faulty core is corrupted, and we measure latency from that point. Latency is measured as the total number of instructions from architecture state corruption (either the OS or the App) to detection. The detection latency helps us understand recoverability of the detected faults.

5.3 Fault Diagnosis

Our diagnosis algorithm uses firmware-based control to run the screening and trace generation phases in native mode while emulating the replay phases. Owing to the complexity of the full firmware implementation, we currenlty mimic the functionality of the firmware within our simulation infrastructure.

The checkpointing and rollback/replay required by the diagnosis algorithm are performed using SafetyNet [25]. In order to ensure determinism during diagnosis, we disable asynchronous interrupts during trace generation and replay phases. Cross calls between cores are, however, always serviced and cannot be disabled in the SPARC V9 architecture. On such interrupts, we abort and restart the current diagnosis phase (trace generation or replay) to ensure determinism.

In the screening phase, permanent faults are expected to cause symptoms and trigger trace generation. However, some permanent faults may not throw symptoms when screened (0.9% in our experiments) due to non-determinism in both the microarchitecture (e.g., instruction scheduling) and the software execution (no deterministic replay in this phase). We diagnose these faults as transients that are recoverable with rollbacks, and do not trigger trace generation for these faults.

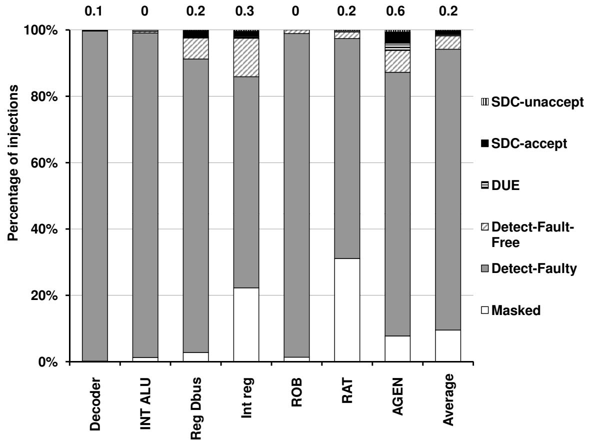


Figure 5: SDC rate for mSWAT for permanent faults in various structures. The symptom detectors result in very low SDC rates while incurring near-zero hardware overheads and therefore are effective even for multithreaded workloads.

For trace generation and replay, we study the iterative diagnosis approach with an iteration length of 100K and 1M instructions. We do not model the cache/memory traffic or latency generated by the read or write of the trace buffers. Although, this does not affect the correctness of the implemented diagnosis, it affects the reported performance evaluation. The performance evaluation is further affected by our mimicking of the firmware within our simulator and running replays in native execution mode. This under-estimates the overhead of a firmware based approach by about 10-20X (documented overheads of state-of-the-art emulators).

We evaluate mSWAT diagnosis using two metrics – diagnosability, and diagnosis latency. *Diagnosability* is the fraction of detected faults for which the faulty core is correctly diagnosed. *Diagnosis latency* is the number of cycles from the start of screening phase until the end of the last replay phase (first or second). We conservatively multiply the latency of the replay phases by 20X to account for the slowdown due to emulation. In order to understand the incurred hardware overheads, we also measure the sizes of the loadLog and compareLog collected in the trace generation phase.

6. RESULTS

6.1 Fault Detection

6.1.1 SDC Rate

Figure 5 shows the SDC rate from our symptom detectors for permanent faults in different structures. For each case, the figure shows the percentage of faults that are masked by both the architecture and application. Faults detected within 10M instructions are classified into those detected by a symptom in the faulty core (*Detect-Faulty*) and those detected by a symptom in a fault-free core (*Detect-Fault-Free*). Faults detected beyond 10M instructions are classified as *DUEs*. The undetected faults are further classified into *SDC-acceptable* and *SDC-unacceptable*. The numbers on top of each bar show the SDC-unacceptable rate of the symptom detectors for faults in that structure.

The symptom detectors perform very well in this new environ-

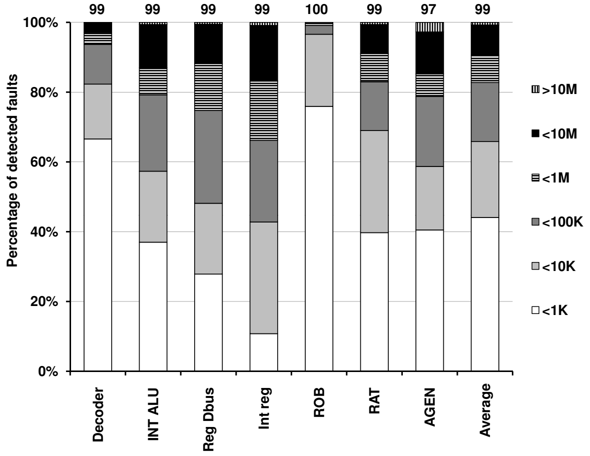


Figure 6: Breakdown of detection latency for detected permanent faults. A large fraction of the faults are detected within 10M instructions, and can be recovered with support for hardware checkpointing.

ment, resulting, in a low average SDC rate of 0.2%. Further, the SDC rate is fairly uniformly low across the different structures. This shows the efficacy of symptom detection to detect faults in multicore systems running multithreaded workloads.

Figure 5 also shows the effect of cross-core fault propagation – 4.5% of the detected permanent faults are detected on fault-free cores. This stresses the importance of the diagnosis algorithm to handle such cross-core fault propagation.

6.1.2 Detection Latency

Figure 6 gives the breakdown of the detection latency for the detected permanent faults. The numbers on the top of each bar shows the percentage of the faults that were detected within 10M instructions. For each structure, the bars are divided into several latency stacks from $< 1K$ to $> 10M$. The result aggregated across all structures is also shown.

From the figure we see that, on an average, nearly all permanent faults (99%) are detected within 10M instructions. Hardware techniques such as ReVive [21] and SafetyNet [25] can handle such latencies for recovery, making these faults recoverable using hardware checkpointing (10 million instructions corresponds to about 10ms on a 1 GHz processor, assuming an IPC of 1). However, system I/O would have to be buffered and delayed for this interval, potentially impacting performance [16].

6.1.3 Transient Faults

Of the injected transient faults, 91% were masked (85% are architecturally masked and 6% are masked by the application). Only 0.5% of the injections faults result in unacceptable SDCs. 73% of the detected permanent faults are detected within 10M instructions. These results are consistent with previous findings for single-threaded workloads [10, 12, 30].

6.2 Fault Diagnosis

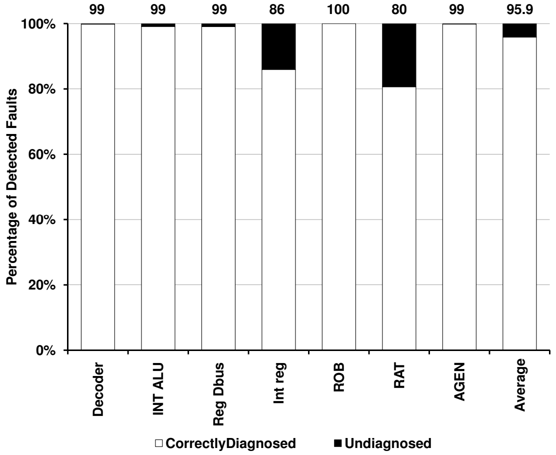


Figure 7: Diagnosability of detected faults that throw symptoms in the screening phase. mSWAT successfully diagnoses 95.9% of such faults, and in particular, all faults that cause symptoms from fault-free cores.

Diagnosability: Figure 7 shows the diagnosability for permanent faults that throw a symptom in the screening phase (99.1% of the faults show a symptom during screening phase) across different microarchitectural structures. Each such fault is diagnosed using the iterative diagnosis algorithm with iteration length of 100K instructions, until either the fault is successfully diagnosed (*Correctly Diagnosed*) or a threshold of 20 million instructions is reached. In the latter case, the fault is not diagnosed (*Undiagnosed*). The numbers on top of each bar show the diagnosability for faults detected in that microarchitecture structure.

Figure 7 shows that 95.9% of the faults subjected to diagnosis are successfully diagnosed. In particular, all the faults that were detected on fault-free cores (4.5% of the detected cases from Figure 5) were successfully diagnosed, demonstrating the ability of mSWAT to diagnose permanent faults in multicore systems.

Our iterative diagnosis algorithm approach, however, does not diagnose 4.1% of the faults – most of these occur in the Integer Register and the RAT. We investigated whether our design choices of storing only parity for load addresses in the compareLog and fixing the iteration size at 100K instructions led to these undiagnosed cases. When comparing all the bits of the load address, instead of the parity, diagnosability improved by a mere 0.03%. Increasing the iteration length to 1M instructions increases diagnosability by only 0.1%. The fundamental reason for these undiagnosed faults is the difference in fault activation between the screening, trace generation, and replay phases. This is in turn due to non-determinism induced from the initial microarchitecture states being different between the screening and replay phases. In particular, 88% of these undiagnosed faults never activate the fault in diagnosis, resulting in no divergence. While the remaining 12% activate the fault during diagnosis, these faults are not diagnosed due to inherent microarchitecture-level non-determinism (due to differences in scheduling, physical register allocation, etc.) that prevents divergences during replay.

Diagnosis Latency: Figure 8 categorizes the diagnosis latency of the faults that are successfully diagnosed into various bins of 100 thousand to 1 billion cycles. The number on the top of each bar shows the percentage of diagnosed cases with latency less than 10 million cycles.

From this figure, we see that 80% of diagnosed cases have latency within 1 million cycles and 98% within 10 million cycles, latencies that are invisible to end users (<10ms in a 1GHz processor assuming an IPC of 1). Further, 93% of these faults were successfully diagnosed within 1 iteration while 98.5% took 10 iterations (results not shown here). This shows that the iterative approach sig-

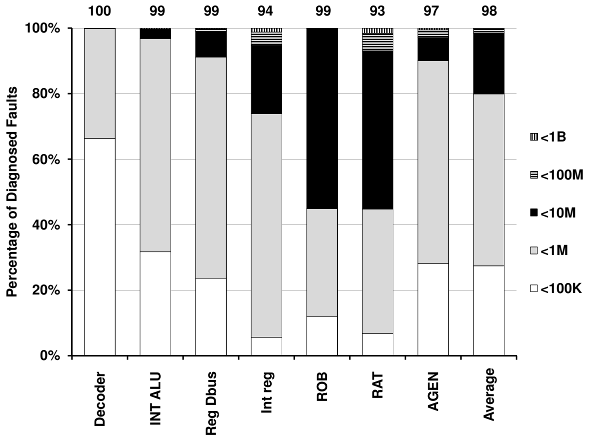


Figure 8: Breakdown of diagnosis latency for diagnosed permanent faults. 98% of the faults are diagnosed within 10M cycles, resulting in low system down time.

nificantly reduces diagnosis latency, by avoiding logging the entire execution before initiating replay.

For 1.5% of the diagnosed faults, the diagnosis latency is larger than 10 million cycles (<100M and <1B in Figure 8). Since diagnosis is a rare process, even these longer latencies are acceptable, albeit undesirable.

Log Size: Figures 9(a) and 9(b) show the sizes of the loadLog and compareLog recorded in trace generation for the diagnosed faults. Since the log size depends on individual applications, the figures present the results on a per-application basis.

We observe that, on average, 96% of the cases require less than 200KB of loadLog and less than 200KB of compareLog during diagnosis. These logs can be easily accommodated in the L2 or L3 caches. In contrast to previous methods that performed deterministic replay of multicore systems, these logs are orders of magnitude smaller (FDR proposed logs in the order of megabytes [31]), require less complex hardware (compared to BugNet [18], our scheme does not require the complexities of memory-race-buffer, dictionaries, etc.), and works with existing memory subsystems (Rerun [7] and DeLorean [15] require changes in the memory subsystem to optimize their hardware overheads from logging).

The results presented in this section show that mSWAT can achieve high detection coverage and low SDC rate while incuring acceptable hardware overheads for fault diagnosis, making it widely applicable for faults even in commodity systems.

7. CONCLUSIONS AND FUTURE WORK

This paper addresses the problem of reliability that is a serious threat to the current computer industry. While recent advances have embraced low-cost reliability solutions as a replacement for traditional high-cost full redundancy techniques, they have focused on single threaded workloads running on a single core. Our focus here is on studying faults in the emerging multicore systems running multithreaded workloads.

In this paper, we propose mSWAT, the first full system reliability solution for multithreaded workloads that uses high-level near-zero overhead symptom detectors to detect faults, and a novel, but more expensive, diagnosis mechanism after a fault is detected. The detectors achieve a low SDC rate for both permanent and transient hardware faults in multicore processors running multithreaded workloads. The diagnosis module, with no prior knowledge of a fault-free core, successfully diagnoses over 95% of the detected permanent faults at low hardware overheads. Further, mSWAT is controlled by a thin firmware layer that requires minimal hardware support, making it widely deployable even in commodity systems. Although the diagnosis algorithm is presented in the context of SWAT, it can be used in combination with other high-level detection mechanisms that may allow a fault to escape the faulty core.

There are many avenues for future work with mSWAT. The detectors can be augmented (e.g., OS and the application-level detectors [23]) to further improve SDC rate and latency. We also need to fully implement the mSWAT firmware and evaluate it under more accurate gate-level fault models for permanent faults. Finally, mSWAT currently focuses on in-core faults and needs extensions to handle faults in off-core components such as the I/O controller, memory sub-system, etc.

References

[1] D. Bernick et al. NonStop Advanced Architecture. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[2] S. Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.

[3] F. A. Bower, D. Sorin, and S. Ozev. Online Diagnosis of Hard Faults in Microprocessors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(2), 2007.

[4] M. Dimitrov and H. Zhou. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *Proceedings of the International Conference on Parallel Archtectures and Compilation Techniques (PACT)*, 2007.

[5] O. Goloubeva, M. Rebaudengo, M. S. Reonda, and M. Violante. Soft-Error Detection Using Control Flow Assertions. In *International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.

[6] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The StageNet Fabric for Constructing Reslilient Multicore Systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2008.

[7] D. R. Hower and M. D. Hill. Rerun: Exploiting Episodes for Lightweight Memory Race Ordering. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008.

[8] M.-L. Li, P. Ramachandran, R. Karpuzcu, S. K. S. Hari, and S. Adve. Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[9] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[10] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. Adve, V. Adve, and Y. Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

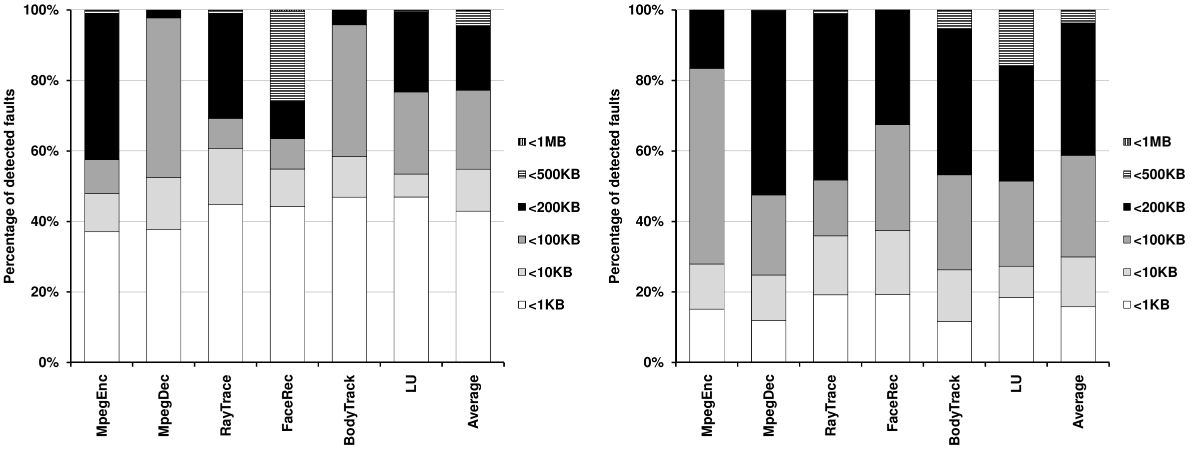
[11] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[12] G. Lyle, S. Cheny, K. Pattabiraman, Z. Kalbarczyk, and R. Iyer. An End-to-end Approach for the Automatic Derivation of Application-Aware Error Detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2009.

[13] C. Mauer, M. Hill, and D. Wood. Full-System Timing-First Simulation. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 2002.

[14] A. Meixner, M. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007.

[15] P. Montesinos, L. Ceze, and J. Torellas. DeLorean: Recording and Deterministically Replaying Shared Memory Multiprocessor Execu-



(a) Size of loadLog

(b) Size of compareLog

Figure 9: The size of the loadLog and compareLog used by our diagnosis algorithm. Over 96% fo the diagnosed faults require less than 200KB of logs per core, which can fit in the L2/L3 cache of modern processors.

tion Efficiently. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2008.

[16] J. Nakano, P. Montesinos, K. Gharacorloo, and J. Torrellas. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2006.

[17] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer. An Architectural Framework for Detecting Process Hangs/Crashes. In *Proceedings of European Dependable Computing Conference (EDCC)*, 2005.

[18] S. Narayanasamy, G. Pokam, and B. Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.

[19] K. Pattabiraman, G. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *European Dependable Computing Conference*, 2006.

[20] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, and T. Austin. CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework. In *IEEE International Conference on Computer Design*, September 2008.

[21] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.

[22] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. Perturbation-based Fault Screening. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[23] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2008.

[24] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[25] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.

[26] L. Spainhower et al. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. In *IBM Journal of R&D*, September/November 1999.

[27] J. Srinivasan, S. Adve, P. Bose, and J. A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.

[28] R. Venkatasubramanian, J. Hayes, and B. Murray. Low-Cost On-Line Fault Detection Using Control Flow Assertions. In *Proceedings of the International Online Test Symposium*, 2003.

[29] Virtutech. Simics Full System Simulator. Website, 2006. `http://www.simics.net`.

[30] N. Wang and S. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3), July-Sept 2006.

[31] M. Xu, R. Bodik, and M. Hill. A "Flight Data Recorder" for Enabling Full-system Multiprocessor Deterministic Replay. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2003.