LOW-COST HARDWARE FAULT DETECTION AND DIAGNOSIS FOR MULTICORE SYSTEMS RUNNING MULTITHREADED WORKLOADS

BY

SIVA KUMAR SASTRY HARI

B.Tech., Indian Institute of Technology, Madras, 2007

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Adviser:

Professor Sarita V. Adve

ABSTRACT

Continued device scaling is resulting in smaller devices that are increasingly vulnerable to errors from various sources, e.g., wear-out and high energy particle strikes. As this reliability threat grows, future shipped hardware will likely fail due to in-the-field hardware faults. A comprehensive reliability solution should detect the fault, diagnose the source of it, and recover the correct execution. Traditional redundancy-based reliability solutions that handle these faults are too expensive for mainstream computing. A promising approach is using software-level symptoms to detect hardware faults. Specifically, the SWAT project [11] has proposed a set of always-on monitors that perform such detections at very low cost. In the rare event of a fault, a more expensive diagnosis mechanism is invoked alongside a checkpoint/replay-based recovery procedure.

Previous studies, however, were in the context of single-threaded applications on uniprocessors, and their applicability in multicore systems is unclear. This thesis provides detection and diagnosis mechanisms for hardware faults in multicore systems running multithreaded applications. For detection, we augmented the SWAT symptoms with multicore counterparts. These resulted in a high coverage of 98.8% for permanent faults, with a low 0.8% silent data corruption (SDC) rate. We also show that these symptoms are effective for transient faults. These results demonstrate the applicability of symptom-based detection for faults in multicore systems running multithreaded workloads.

Permanent faults require a diagnosis mechanism, unlike transient faults. In multicore systems, a fault in a core may escape to a fault-free core, and the latter may result in a symptom. This makes permanent fault diagnosis a challenge. We propose a novel mechanism that identifies the faulty core, with near-zero performance overhead in the fault-free case. Our diagnosis mechanism replays the execution from each core on two other cores and compares the executions. A mismatch in the executions results in identification of the faulty core. Our results show that the proposed diagnosis technique successfully diagnoses 95.6% of the detected faults. We also show that 96% of those cases

were diagnosed within 1 million cycles. We achieve such a high coverage with low area overhead of 2KB per core, and with minimal changes to the processor design. Once the faulty core is identified, we rely on previous work to diagnose the faulty microarchitectural unit.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser, Prof. Sarita Adve. This thesis wouldn't have been possible without her encouragement and advice. I am greatly indebted to her for the continuous motivation and guidance she provided. I also want to thank her for providing me the opportunity to work on such an interesting and challenging problem.

Previous SWAT framework, developed by Alex and Pradeep, forms the basis of my research. I sincerely thank them not only for providing me with an excellent framework to work on, but also for their incredible help during the project. I thank Byn for his valuable assistance in the project and for providing good company in the office, especially during evenings and weekends. I would also like to thank all my officemates for creating an excellent work environment.

I thank my parents, Ramalinga Murty and Prabhavati, who have always taken interest and pride in every activity I undertook. I cannot thank my brother, Sreepati, enough for encouraging me to pursue my interests and for being supportive throughout.

I also thank my friends in UIUC who have accompanied me in the journey of my graduate life. I thank them all for the incredible amount of selfless support they provided. I want to thank Sandeep, Sreekanth, Varun, Siva, Nikhil, Sarath, Meghna, Khan, Gudla, Deepti, Ram, Aftab and Country for all the fun time.

I thank the Department of Computer Science at UIUC for presenting me with the opportunity to pursue my graduate studies. Last but not the least, I thank the university for providing the state-of-the art facilities.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Driven by Moore's Law, continuous device scaling has provided ever increasing system integration. However, the decrease in device size also makes future hardware susceptible to faults due to various phenomena such as high energy particle strikes, aging or wear-out, design defects, infant mortality due to insufficient burn-in, and so on [4]. As a result, in-the-field hardware reliability is a growing concern. 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 [2, 25, 35]. In a recent workshop, an industry panel converged on 10% area overhead target to handle all possible sources of error on chip [28].

Two high-level observations, made by [11], drive our work. (1) The hardware reliability solution should handle only those faults that propagate to higher levels of the system and affect the software execution. (2) Despite the growing reliability threat, fault-free operation remains the common case. Hence, we must optimize for the fault-free common case and keep the cost of the fault detection mechanism low.

These observations motivate a fault detection mechanism, that watches 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 fault mechanism. Such an approach treats hardware faults analogous to software bugs, potentially leveraging solutions for software reliability to further amortize overhead. A large body of prior research has explored various forms of symptom-based detection. Most of this work has focused on transient fault detection [7, 8, 17, 19, 22, 37, 39], which requires no additional provision for diagnosis. Other studies explore permanent fault detection, without exploring diagnosis [15].

Unlike transient faults, permanent faults require a diagnosis mechanism. A reliable system should be able to diagnose the source of the failure, repair/reconfigure the faulty unit, and recover from it. Therefore, the diagnosis module should identify the source of the failure by identifying the faulty hardware component (at core level or at microarchitecture level).

To the best of our knowledge, the SWAT (SoftWare Anomaly Treatment) project [10, 11, 26] provides the most comprehensive exploration of the above approach to date, incorporating methods for detection and high-level diagnosis of both permanent and transient faults. This work has shown that a small set of simple and high-level detectors (Chapter 2) can provide very high detection coverage at negligible cost [11, 26]. It has also shown a synergistic diagnosis algorithm to isolate the case of permanent faults and determine which microarchitecture-level component is faulty [10].

A current limitation of SWAT, and all of the above cited work, is that it assumes a single-threaded application running on a single core. However, for the foreseeable future, it is expected that multicore hardware and parallel software will be more prevalent. For techniques such as SWAT to have a practical impact, they must be demonstrated to work on applications running on multicore systems.

A key challenge is that a fault in a core may escape to a fault-free core (we call such an event as *cross-core fault propagation*), and the latter may result in a symptom that detects the fault. This makes diagnosis hard, as one can no longer assume that the symptom-causing core is faulty (this assumption is valid for single-core executions and is correctly exploited by SWAT). Furthermore, in a multicore environment where the fault is detected, there is no known good core. The existing SWAT system relies on isolating such a core and on performing deterministic replay for diagnosing the fault at microarchitecture-level granularity. This isolation of the fault-free core, from all the available cores, is another key challenge that the multicore diagnosis algorithm would have to address.

This thesis investigates the use of the SWAT approach to detect hardware faults and proposes a novel approach to diagnose permanent faults in multicore hardware running multithreaded applications. In particular, we make the following contributions.

- **Detection:** We extend the existing SWAT detectors for multicore processors, and evaluate their effectiveness to detect hardware faults with multithreaded applications running on a multi-core processor with a real operating system. The focus of our work is on permanent faults, but we also show that this approach is effective for transient faults. The augmented SWAT detectors provide a high fault detection coverage of 98.8%, with a low Silent Data Corruption (SDC) rate of 0.84%, for permanent faults. The same detectors also provide high coverage for transient faults, with low SDC rate of 0.5%. These results show that detecting anomalous software behavior as symptoms of underlying hardware faults also works in multicore environments. Further, 12.3% of the injected permanent faults cause symptoms from fault-free cores, confirming that the cross-core fault propagation is prominent in multicore systems. Therefore, the diagnosis mechanism for a multicore system should be able to diagnose faults that cause symptoms on fault-free cores.

- **Diagnosis:** We propose a novel algorithm that distinguishes between various fault sources (transients, software bugs, permanent hardware bugs) when a fault is detected in a multicore system. Most importantly, for permanent hardware faults, the algorithm successfully identifies the faulty core. The algorithm deterministically replays the fault activating execution from each core on two other cores. The algorithm uses a checkpoint and a buffer of the load values from each core to perform deterministic rollback and replay of execution on each core. It employs a voting mechanism to compare the three executions during replay. Hence, a mismatch in the replay would result in a diagnosis decision, identifying a faulty core. Out of the $7,565$ detected permanent faults that were subject to diagnosis, this algorithm successfully diagnosed 95.6% of the faults. In particular, all the faults that resulted in symptoms in fault-free cores are successfully diagnosed by the diagnosis algorithm. Additionally, our results show that 96% of the diagnosed cases have diagnosis latency within 1 million cycles (equivalent to 1ms in a 1 GHz processor). We minimize the area requirement of our technique to mere 2KB per core, with minimal changes to the processor design.

To the best of our knowledge, this is the first work that provides a comprehensive solution to 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 rare case. Fault-free operation, which remains the common case, continues to see near zero overhead.

CHAPTER 2

RELATED WORK AND BACKGROUND

Reliable system design has been a prominent area of research in computer architecture. There is a large body literature available on designing reliable architectures. HP NonStop [2] and IBM S/390 G5 [35] are known to provide high reliability through redundant hardware. Austin proposed DIVA [1] which checks every retiring instruction for errors using an efficient checker processor. Another approach uses time redundancy for transient fault tolerance by replicating the program execution [25]. These approaches are expensive in terms of area, power and performance. There has been a lot of work on partial redundant threading architectures [9, 27, 31, 33, 36]. Most of these approaches still have high performance penalty for the coverage they provide.

There have been approaches that perform periodic on-line testing of the structures in the microprocessor [6, 30]. If a unit is found to be faulty, it will be repaired/reconfigured and execution will continue after rolling back to a pristine checkpoint. These approaches have performance overhead in the fault-free execution, which is undesirable.

The above mentioned approaches have performance overhead in the fault-free common cases, increase wear-out, and increase power consumption . On the other hand, the symptom based fault detection and diagnosis mechanisms have almost zero overhead in the fault-free execution and they do not increase wear-out (no additional computation is required). As mentioned in Chapter 1, recent research has focused on symptom-based fault detection mechanisms. To the best of our knowledge, the SWAT project provides the most comprehensive reliability solution based on symptom-based detection techniques for permanent faults. Since our work is based on the SWAT approach, we provide a detailed summary the SWAT project.

2.1 SWAT: SoftWare Anomaly Treatment

The SWAT project investigates how future hardware can be protected from in-the-field failures with very little overhead in area, power, and performance. As mentioned in Chapter 1, the key observations that drive the design of the SWAT system are (1) hardware faults need to be handled by the reliability solution only if they manifest and appear as software bugs, and (2) the fault-free operation remains the the common case. Based on these two observations, SWAT uses low overhead detectors of software anomalies for detecting hardware faults. Since the diagnosis is rarely invoked, relatively high-cost for diagnosis is acceptable.

2.1.1 Detection

SWAT detects hardware faults by employing very low-cost hardware monitors that detect anomalous software behavior. Li et al. first proposed the following simple detectors that require very little hardware support and no software support [11]

1. *Fatal-Traps*: Traps such as division by zero and misaligned memory access (in SPARC) are not thrown in normal execution and are indicators of anomalous software behavior. These are used as zero-cost detectors – these traps transfer control to the SWAT firmware that then invokes diagnosis.

2. *Hangs*: Application and system hangs are symptoms of anomalous software behavior. SWAT identifies them using a simple hardware hang detector that monitors the frequency of branch instructions.

3. *High-OS*: Operating Systems are designed to incur as little performance overhead on the application as possible. Thus, abnormally high amount of time (more than 10,000 instructions) spent contiguously in the OS, except for system calls and interrupts, is considered an anomaly. This detector considers excessive amounts of contiguous OS activity (>50k contiguous privileged instructions) as a symptom of a fault. Existing performance counters can trivially identify such scenarios.

Once a symptom is detected, control is transferred to the firmware, which initiates the diagnosis procedure. If the diagnosis procedure (section 2.1.2) determines that the symptom was not caused due to a hardware fault, the symptom is deemed a false positive of the hardware fault. Fatal-traps are not prone to false positives, where as, hangs and high-OS are prone to false positives, as they are based on heuristics. On identifying a symptom as false positive, the diagnosis procedure may adjust the thresholds of these detectors. In general, there is a trade-off between how aggressive these symptom detectors can get and the rate of false positives.

Sahoo et al. later developed detectors that mined likely invariants from the application and inserted application code to monitor the violation of these invariants as symptoms of a hard fault [26]. This detector further reduced the Silent Data Corruption (SDC) rate, improving the coverage of SWAT. In addition, these invariants incur low performance overheads in fault-free execution (5% on x86 machines), rendering them effective monitors for faults.

We found that the hardware-only detectors from [11] with modest extensions worked very well for our experiments. We do not explore Sahoo et al.'s detectors here, since they require application binary modifications. Extending such detectors to multithreaded applications is a promising area for future work.

2.1.2 Diagnosis

Since software bugs, transient faults, and permanent faults may all result in software-level symptoms, the diagnosis module of SWAT should distinguish the source of the fault. However, diagnosis is rarely invoked; it can incur higher overheads. SWAT assumes a single threaded application running on a single core, but assumes the availability of another fault-free core to help with diagnosis.

The SWAT diagnosis algorithm (Figure 2.1) relies on repeated rollbacks/replays on the faulty core (where the symptom is invoked) and another core (assumed to be fault-free) to distinguish between the above three types of faults. The algorithm works as follows. (1) The execution is replayed on the same core and if the symptom does not recur, a transient fault is diagnosed and the execution continues. (2) If the symptom recurs, then a fault-free core executes the same trace. If the symptom 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, a permanent fault is diagnosed on the original core.

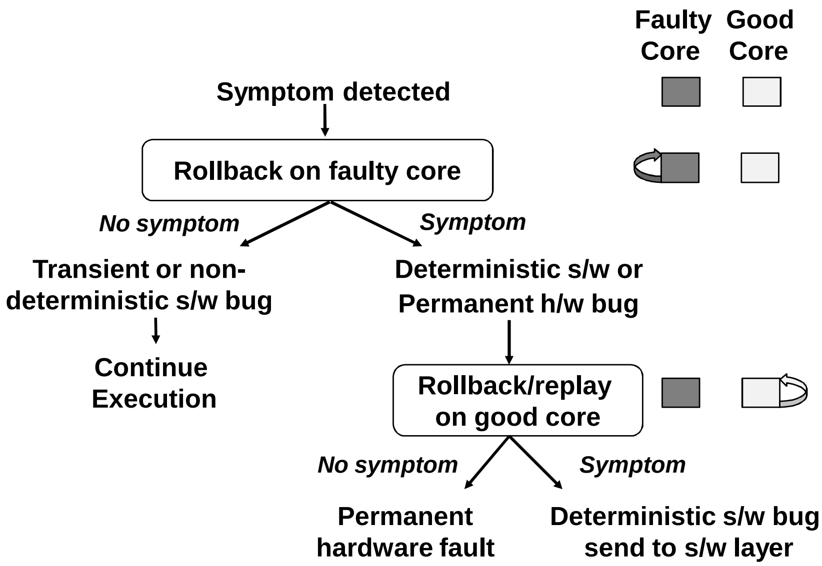


Figure 2.1 SWAT diagnosis algorithm.

Once a permanent fault is identified in a core, Trace-Based Fault Diagnosis (TBFD) is invoked to diagnose the permanent fault at the microarchitecture-level granularity. A successful fine-grained diagnosis would repair (or reconfigure) only the faulty unit, without rendering the core useless. TBFD employs a synthesized Dual-Modular Redundancy (DMR) approach to compare and analyze the instruction traces from the faulty and fault-free core to pinpoint the faulty microarchitectural unit. TBFD relies on clues from the divergences in these traces as manifestations of faults in the microarchitecture-level structures used. Using these clues, TBFD identifies the faulty microarchitecture-level unit. It has been empirically shown that TBFD successfully diagnoses 98% of the detected faults [10], making it a highly effective diagnosis method.

As discussed, we cannot directly apply SWAT's diagnosis mechanism (TBFD) for a multicore execution because the symptom invoking core may not be the faulty core and we do not know which core is fault-free. This thesis presents a novel technique to identify the faulty core in a multicore system. Once the faulty core is identified, SWAT's diagnosis (TBFD) can be applied to identify the faulty microarchitecural unit. We also provide the algorithm to distinguish between transient faults, software bugs and permanent faults in multicore systems.

2.2 Checkpointing Mechanism

Since our technique requires rolling back and replaying the execution, we need a checkpointing mechanism. Several techniques have been proposed for checkpointing for the purpose of recovery from hardware faults, such as SafetyNet [34] and ReVive [21]. We are using a scheme similar to SafetyNet to checkpoint the memory state.

2.3 Deterministic Replay

As discussed later in Chapter 4, our diagnosis algorithm requires the ability to deterministically replay each core's execution in isolation from the other cores. There has been prior work on replaying multithreaded workloads in the context of software bug detection, such as BugNet [18] and FDR [40]. In particular, BugNet deterministically replays the program execution to find application level bugs. BugNet records the load memory accesses to deterministically relay the execution. It optimized the data recording mechanism by logging only the first loads to a location, reducing the log size significantly.

We leverage the idea of logging the load values to facilitate deterministic replay, as proposed in BugNet. There are, nevertheless, significant differences in our work. Specifically, we replay each core's execution in parallel with other cores, some of these cores may be running the same "trace". This makes the values from memory untrustable. Therefore, we cannot use the optimization in BugNet of logging just the first loads to a location; we need to log all loads. However, load values show frequent value locality (shown in [41]). Hence, we can use the dictionary-based compression technique for storing the load values, similar to the one that is used in BugNet.

CHAPTER 3

MULTICORE FAULT DETECTION

In order to detect faults in multicore systems, we first use the SWAT hardware-only symptom detectors, namely, Fatal-Traps, Hangs and High-OS. These are described in Chapter 2.1.1 From our experiments, we found two additional symptoms to be valuable – *Panic* and *No Forward Progress*.

1. **Panic**: When an operating system detects that it is in an invalid state, a panic is initiated in order to minimize potential damage to the user data and to facilitate debugging. System enters an invalid state due to fatal operations such as a read to an invalid or non-permitted memory address from OS. The equivalent in Microsoft Windows OS is the "Blue screen of Death" and in Unix is "kernel panic". In most modern operating systems, this is a centralized routine, whose location is commonly disclosed for the purpose of reporting bugs in the kernel. Thus, identifying this symptom requires minimal support from the OS, which already exists. The faults that are detected by Panic can also be detected by High-OS and/or Hangs, but, with much higher detection latencies.

2. **No-Forward-Progress**: In multithreaded workloads, a fault may result in the lack of forward progress in the application, as the threads may wait on each other indefinitely. In this period, none of the cores retire application-level instructions. We thus detect No-Forward-Progress in the application by monitoring for excessively contiguous OS activity on each core. High-OS from SWAT, was monitoring anomalous operating system activity on each core independently, whereas, No-Forward-Progress monitors the OS activity in all the cores simultaneously. Therefore, No-Forward-Progress is capable of detecting faults that cause live-lock in the operating system.

With the addition of these two symptoms to SWAT, we show that the SWAT approach can be effective for multicore systems.

Since the above symptoms gave exceptionally high coverage, we did not pursue research in developing new symptoms. The likely program invariants [26] can also be applied as a detector. This detector can improve detection latencies and reduce silent data corruptions. We leave this for future work.

CHAPTER 4

MULTICORE FAULT DIAGNOSIS

Since we detect faults by watching for symptoms of anomalous software behavior, any underlying fault can manifest as a symptom. The diagnosis procedure must first determine whether the symptom was caused due to a software bug, a transient hardware fault, or a permanent fault – each category requires a different subsequent action.

In particular, for a permanent fault, the diagnosis procedure must determine which core is faulty and possibly, which component in the core is faulty. Depending on the available options, the faulty component can be reconfigured. In this work, we only consider faults in the core. We assume a single core fault model; i.e., at most one core is faulty. Since the SWAT symptom may have been detected on a fault-free core due to fault propagation across cores (either through the application or the OS), our diagnosis solution must first determine which core is faulty. Once the faulty core is detected, the diagnosis procedure must then determine which component within the core is faulty, depending on the granularity of the field-reconfigurable unit.

The key insight behind SWAT's diagnosis of single-threaded programs running on a single core, Trace Based Fault Diagnosis (TBFD) [10], is that the execution that generated the symptom can be used as a *test trace* to repeatedly activate any faults present to incrementally perform diagnosis. TBFD repeatedly replays this execution on both the faulty core and a good core. Depending on whether a symptom recurs on either core, the fault can be diagnosed as a software bug or a hardware permanent or transient fault. For permanent faults, SWAT inexpensively synthesizes DMR between the two (faulty and good) cores – a careful comparison of the execution traces on the two cores using the TBFD algorithm determines which microarchitectural component is faulty (see section 2.1.2 for more details). Note that TBFD assumes that the faulty core is identified and a known good core is available.

A naive extension of SWAT's diagnosis algorithm 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 and replay it on a set of N good cores. Thus, a naive algorithm must assume that N known good cores are available, making it too expensive.

A simple optimization can use one spare core, that is known to be good, (a total of $N+1$ cores) to do N replays of the multithreaded application with different subsets of N cores to identify the faulty core. The deterministic replay of the multithreaded application can be done using techniques such as BugNet [18] and Flight Data Recorder [40]. This is not a scalable solution, however, because it could take up to N replays to identify the faulty core, further it also requires a known good spare core just for this purpose.

We propose an algorithm that can diagnose a faulty core in a system of N cores, where $N \geq 3$, with a maximum of 5 (3 + $mod(n, 3)$) replays. We also eliminate the requirement of a spare core. Thus, our algorithm is scalable. It does not require spares for diagnosis and eliminates single point of failure (the known good spare good in the naive algorithm).

4.1 Diagnosis Algorithm

Once a symptom is detected, all the cores are rolled back to the previous checkpoints, and the diagnosis algorithm is run to identify the faulty core. We assume the availability of checkpoint/replay mechanism in our system. Our algorithm (Figure 4.1) performs fault diagnosis in three phases, described as follows:

1. **Replay phase**: All the threads are first replayed from the rolled-back checkpoint. If no symptom occurs during replay, we diagnose the fault as a transient hardware fault or a non-deterministic software bug and continue execution. If a symptom occurs, then it is caused either by a permanent fault or a software bug. In this case, we move on to the next phase of diagnosis.

2. **Logging phase**: The execution is restored to the previous checkpoint and the multithreaded workload is replayed again. This time, each core collects the value and address of each of its retiring load instructions in per-core Load-Log Buffer (LLB). These logs are then used to

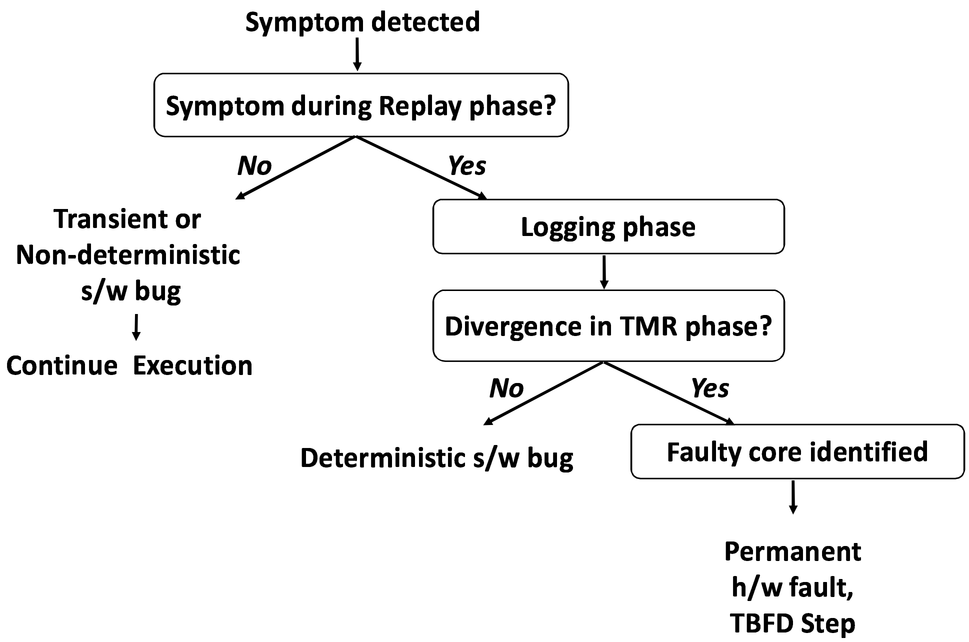


Figure 4.1 The diagnosis decision tree

deterministically replay the execution of each core in isolation in the next phase.

3. **TMR phase**: In this phase, the execution from each core is run on thee different cores. The three executions are compared through a voting mechanism. Effectively, we synthesize Triple-Modular Redundancy (TMR) for this phase. This phase performs TMR execution for each of N cores, in the following way:

   - The given core's state is restored to the previous checkpoint.
   - The same checkpoint and the core's LLB from Logging phase are loaded on two other cores.
   - The three cores deterministically replay the execution of the given core by using load values from the LLB buffer.
   - The execution of the three cores is compared, and a divergent core is declared to be faulty.

   If there is no divergence in any of the N cores' TMR executions, then a software bug is assumed and control is returned to the appropriate software layer.

   This phase can be run in parallel for $N/3$ cores at a time.

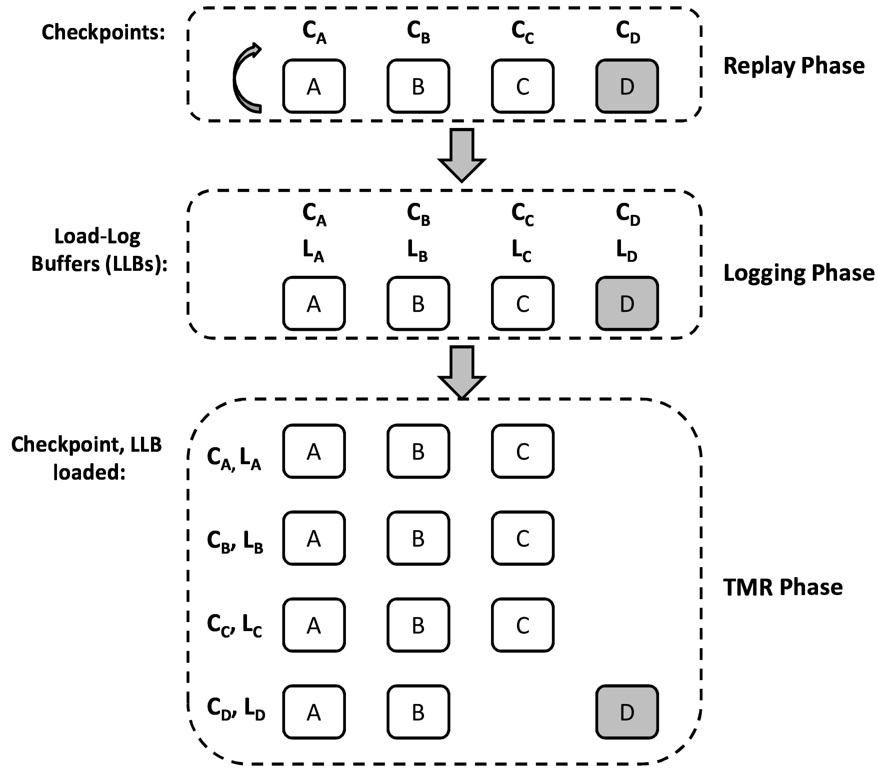


Figure 4.2 Diagnosis mechanism

Figure 4.2 shows an example of the described diagnosis mechanism. Here, a symptom is detected in core D, marked gray. After detection, all the cores are rolled back for the *Replay phase*. In this case, the symptom is detected again in core D and that triggers the *Logging phase*. In Logging phase, all the cores record their load values and addresses in their respective LLBs (e.g., in L_A for core A, in Figure 4.2). Again, the symptom is detected in the Logging phase which triggers the *TMR phase*. The checkpoint and LLB of core A (C_A and L_A) are loaded on cores A, B, and C for TMR execution. These three cores start execution from the checkpoint, reading load values from LLB for a fixed number of instructions. The three executions traces are compared for diagnosis. In this case, there is no divergence, indicating that core A is not faulty. Similarly, the executions for each of the cores B, C and D are checked. In the last step of the TMR phase, a divergence is seen, and core D is identified as the faulty core.

In the following text, we use the term *step* to refer to one reply. For example, one *TMR phase* consists of 4 *TMR steps* (from Figure 4.2).

The key insights behind our technique are as follows:

- We reduce our problem from that of using a test trace of an N core execution to that of using a test trace on a single isolated core. We do this by using an insight from BugNet [18] – a single core's execution from a multicore trace can be replayed in isolation by logging all the values read by the loads in the trace.

- If we assume that at most a single core is faulty, then we can determine the faulty core by synthesizing TMR on our system. We run the isolated single-core trace with the logged load values on every subset of three cores. Any divergence of the outcome within a subset indicates a faulty core.

- Once the faulty core is detected, we can apply SWAT's TBFD between the faulty and another good core, to get diagnosis at the granularity of microarchitecture-level.

The following sections describe the Logging and the TMR phases in more detail.

4.2 Logging Phase

If the *Replay phase* invokes a symptom, the *Logging phase* is performed to create logs to enable isolated deterministic replay for each core. The memory address and memory values of the load instructions from each core are logged into a per-core structure called the Load-Log Buffer (LLB).

These logged <address,value> pairs in the LLB are sufficient to replay the execution of that core in isolation from other cores. (Although storing only the load values in LLB is sufficient for the replay, logging the address helps us in identifying faults in the address. This is explained in more detail in Section 4.3.) Since the sharing between threads in a shared-memory machine happens through memory, logging the loads from each thread is sufficient for deterministic and isolated replay.

This logging continues until either a symptom is detected, or the same number of instructions are committed as in the Replay phase (a symptom may not be thrown in the logging step as the microarchitectural state is not checkpointed, leading to differences in fault activations). Ideally, we would want to log all the load instructions (in LLB) that lead to a symptom. Since the number of instructions that lead to symptom can be large, there are practical limitations on the size of LLB. We discuss a way to handle this issue in the Section 4.4.1, whereas, in Section 4.3 we assume that the LLB is unbounded.

4.3 TMR Phase

The TMR phase is the core of the diagnosis algorithm to identify permanent hardware faults and software bugs. As previously discussed, replay in TMR mode is required in a multicore system in order to identify the faulty core.

Since it is known that the execution from Logging phase has activated the fault, the fundamental idea that is exploited throughout this phase is to replay the execution of a core from the Logging phase in the *same core*. For example, the execution of core A in the Logging phase, should be replayed on core A along with two other cores during TMR phase.

4.3.1 Ensuring Deterministic Replay

After the Logging phase terminates, the cores are rolled back to their pristine checkpoints (processor, and TLB state[1]). The logged values in the LLB then facilitate deterministic, isolated, and mostly parallel replay for each core of the multicore system. To ensure deterministic replay, the values of the retiring load instruction should match the recorded load values from LLB. Feeding the value from the LLB to the load instructions can be done at several stages of the pipeline. We outline two approaches:

- The execution of the load instructions is unaltered. The load instructions read from the memory during execute stage, and if the destination register values match the values from the LLB then the execution continues without interruption (in this approach, the LLB records the destination register values during Logging phase). The values from LLB may not match the values of destination registers of the retiring load instruction, because the execution from other core may have written to the same address. In this case, the destination registers are patched with the values from LLB and the pipeline is squashed, to ensure that the subsequent instructions that use these registers as source will get the correct value. In this approach the design of the LLB remains simple since the LLB can be implemented as a queue. There are two disadvantages to this approach: (1) The diagnosis latency possibly increases if there is repeated squashing. (2) Diagnosability may also be lost because faults in the microarchitecture are cleared on each flush.

[1]TLB misses are frequent in faulty executions [39], and we have also observed that significant number of detections are initiated by TLB misses. Hence, to replay the execution of a TLB miss we checkpoint TLB state.

- The LLB stores the memory values read by the load instructions during the Logging phase. In the TMR phase, the load instructions read from the LLB instead of reading from memory.

  This can be implemented with minor changes to the pipeline. In the TMR phase, the load instructions are given a LLB queue pointer at the decode stage. This pointer is used by the load instruction to obtain the load value in the execute stage, instead of reading from memory. The load instruction successfully retires if the LLB queue pointer matches the top of the LLB queue. If the LLB queue pointer does not match the top of the LLB queue, the pipeline is squashed and the LLB read pointer is reset. In most of the cases, the LLB queue pointer matches the top of the LLB queue. In a rare event of branch mis-prediction, the LLB queue pointer can mismatch the top of the queue, causing a squash.

  The LLB queue pointer is updated in the decode stage with the access size of the load instruction. The subsequent load instructions use the updated LLB queue pointer. In some cases, the access size of the load instructions can not be determined at the decode stage (e.g., for *load alternate* instructions the access size is computed in the execute stage). For this reason, the LLB queue pointer is corrupted and the subsequent load instructions would potentially read wrong values. This is another scenario when the load instructions would be squashed.

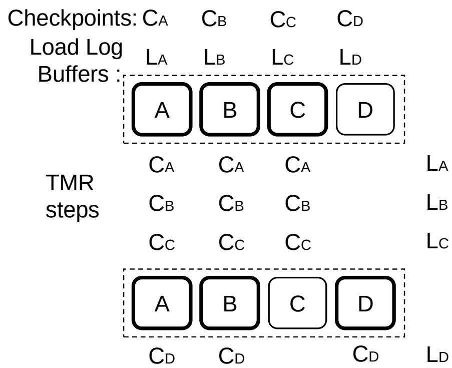
  This approach greatly reduces the number of squashes. We analyze the overhead due to squashing in Chapter 6. A disadvantage of this approach is that the LLB design may get more complicated due to the additional ability to read from an arbitrary index of the LLB queue.

  In our experiments, we use the second approach because we believe it reduces diagnosis latency with minimal added complexity.

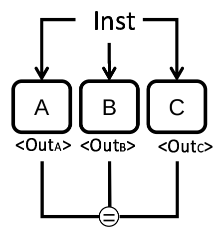
The TMR phase has the capability to reduce the diagnosis latency by intelligently selecting the order of the cores for replay.

4.3.2 TMR Policy

The TMR policy groups the cores into sets of 3, resulting in $N/3$ groups, assuming on total of N cores. Each group performs TMR in parallel, while the remaining one or two cores perform TMR after all the groups have finished.



(a) Example TMR Policy



Inst type	What to compare
Store	Address, data
Branch or Trap	Target PC
Load Miss on LLB	Hit/Miss

(b) Compare points in replay in TMR

Figure 4.3 The TMR phase of the diagnosis algorithm. (a) shows an example TMR policy with a 4-threaded workload on 4 cores, while (b) shows the various points at which the replays in the TMR phase are compared for diagnosis.

In each TMR group, the three cores, A, B, and C, replay the execution of one of the cores, say A, from A's checkpoint driven by the same load values from A's load log buffer (LLB).

Given the checkpoints of A, B, and C, namely, C_A, C_B, and C_C, the cores A, B, and C all execute from C_A, then C_B, and then C_C. This ensures that the faulty core will replay its own faulty execution and thus it will likely activate the fault again, improving the chances of correct diagnosis.

Overall, this results in $3 + 1 = 4$ TMR steps, except for the special case where $N = 5$. 5 TMR steps are required in such case, since the 2 remaining cores cannot be checked in parallel because only one TMR group can be formed at any given time. Figure 4.3(a) shows an example of this grouping and replay in a 4 core system.

While a linear policy of replay (first replay C_A, then C_B, and then C_C on the group with cores A, B, and C) will work, we can use additional optimizations in this selection policy to reduce the number of TMR steps for diagnosis. For example, for programs that have minimal sharing, it is most likely that the symptom causing core is the faulty core. Thus, if the symptom causing thread were first replayed in the TMR step in the group that contains the symptom causing core, the number of replays incurred may be further reduced. However, for programs that have high amount of sharing, other policies may be more effective.

For many-core machines, the TMR policy may choose to replay the execution on the cores that are close to each other to reduce the overhead incurred due to the long latency transfers. For example, in a clustered machine, an intelligent TMR policy would rerun the execution on the cores in the same cluster.

4.3.3 Comparing TMR Executions

In the TMR replay, the threads are run in isolation of each other and of memory. Since the LLB has a log of all the load instructions of that thread, it feeds all the destination registers of the load instructions with the correct values. Stores are not allowed to write their values to memory in order to avoid any memory corruption. This is crucial because we do not checkpoint memory state at the end of the Logging step. We want the memory state to be in tact for the next Logging step to mimic the same behavior as the Replay phase. The replays of the same thread on the three cores are recorded and compared for divergence at the end of each TMR step.

Comparing the traces of all retiring instruction would certainly result in successful diagnosis. However, since unmasked faults either affect data or control values, we need to minimally compare only loads, stores and branches in replay. We present a detailed study in Chapter 6 showing the limitations of comparing only the loads, stores and branches. Further, since the LLB already logs the correct address and data of the load instruction, every load need not be compared. Sufficient diagnosis information is achieved from only those loads that miss in the LLB (explained below). These three criteria form the basis of our comparisons during replay, shown in Figure 4.3(b).

1. *Store instructions:* Address and data values generated by the stores in each core (which are not sent to memory) are logged for comparison. A mismatch in these values signifies the propagation of the fault to a store instruction, leading to diagnosing the core which generated the mismatching store to have a permanent fault.

2. *Branch targets:* Faults propagated to control instructions can be identified by logging and comparing the target PCs of branches. Again, the core that generates the divergent instruction is identified to be faulty.

3. *Loads that miss in the LLB:* An interesting scenario in the TMR step is when a load address in the replay does not match the one in the LLB. Since these indicate some form of erroneous behavior (in either the LLB or in the re-execution), the diagnosis algorithm would successfully diagnose faults in a mismatch event. If these mismatches are not used for comparison, the fault

may be undiagnosable because it may not propagate to stores or branches. For example, if the fault is in the unit that affects the load addresses (address generation unit or integer ALU), then not storing the address in the LLB would lead to a hit, and hence we lose diagnosability. Since the address of the logged load is required to identify a hit or a miss in the LLB, it is necessary to store the load address to identify such faults. Hence, we log the address of the loads in the LLB.

The core that generated the miss is immediately stopped from executing any further instructions, and the diagnosis algorithm waits until all the other cores in this group reach the same load (cores that have already passed these loads can also be stopped as the corresponding load hit in the LLB, which is sufficient information for successful diagnosis). Three possible scenarios of such misses are possible. First, if all the three loads in the replay miss in the LLB, it is because the LLB was incorrectly populated, resulting in diagnosing the core that generated that trace as faulty. Second, if loads from two replays miss in the LLB but the third one hits, this implies that the LLB contains faulty load information and thus the core that logs the data into the LLB is faulty. This is possible only when the core that produced the hitting load is both faulty and the core that initially ran the thread that is being replayed. Hence, this core is then diagnosed as faulty. Finally, if two loads hit, but the third one misses, the core generating the missing load is diagnosed as a faulty core.

There are several ways to collect the execution to be compare at the end of each TMR step during the TMR phase. We outline two approaches:

1. Since diagnosis is not performance critical, we can collect the execution entirely in memory or cache. An efficient alternative would be to collect the execution in a small buffer and flush it periodically to the memory. TBFD [10] uses this approach.

2. Hardware signatures can be exploited for collecting the values from the execution. A bloom filter based hashing function [3] can be used where the comparison of several values can be done in single operation. There have been several techniques that use signatures to disambiguate addresses [5, 42, 29]. Another approach, Fingerprinting [32], uses a fingerprint (computed using a linear block code such as CRC-16) to summarize several instructions' state into a single

value. Using hardware signatures is inexpensive and also reduces the overhead of comparing the entire trace after each TMR step to merely few operations. Hence, we believe the use of signatures can be effective in our technique.

In our experiments, we do not model the collection and comparison of the TMR executions accurately.

Since we have Triple Modular-Redundancy, even the first divergence on any of the above mentioned criteria leads to a successful diagnosis, resulting in a termination of the diagnosis algorithm. Since we have now identified a faulty core and at least one fault-free core is available, fine grained microarchitecture-level diagnosis can be invoked [10]. We, however, do not report this step in our results as we can directly use existing work for this purpose.

4.4 Optimizations to Reduce Hardware Overhead

The previous sections have explained the fundamental concepts used in our diagnosis mechanism, ignoring constraints on area, power, and performance overhead. This section discusses optimizations that can be easily incorporated in present systems with minimal added complexity, and that can reduce the overhead in terms of area, power, and performance.

The idealized technique explain earlier, logs the entire execution (in LLB) till a symptom is seen, requiring large LLB. Hence, reducing the LLB size is crucial for our technique to be practically applicable. To explain the severity of the problem, let us assume a case where the symptom is detected after 1 million instructions (8% of our detected faults have latency higher than 1 million instructions). Also conservatively assume that 25% of the instructions are loads. In this case, the LLB should have $250,000$ entries storing both address and value (64bits each). This is 16MB per core. Hence, reducing the LLB size is crucial for our technique.

4.4.1 Iterative Diagnosis Approach

To reduce the hardware requirement of our technique, we propose an *iterative approach* where the *Logging phase* and the *TMR phase* are replayed repeatedly on short traces until divergence is observed or a predefined maximum number of instructions are executed. In the iterative approach, the decision of when the cores should rollback to start the TMR phase after the Logging phase is made depending on one of the following conditions:

- *LLB is full:* If the LLB fills up, successive load values cannot be logged. Hence, the TMR phase starts.
- *Symptom is detected:* TMR phase starts on seeing a symptom in the Logging phase.
- *Logging threshold is reached:* If the execution exceeds a predefined threshold and no symptom is seen, the diagnosis stops and identifies this case as a non-deterministic software bug. A symptom may not be thrown in the logging step because of microarchitectural non-determinism, leading to differences in fault activations. This is one of the limitations of our technique.

Our iterative approach replays the Logging and TMR phases repeatedly. Each iteration consists of one Logging phase and one TMR phase. At the end of each Logging phase the processor state and the TLB state is checkpointed, to resume the execution in next iteration. We checkpoint TLB state to ensure deterministic replay in the TMR phase. An example run of the iterative mechanism is explained in Figure 4.4. This example has four cores, similar to our previous examples. The iterative diagnosis mechanism proceeds as follows:

All the cores are rolled back to the pristine checkpoints after the replay phase and the Logging phase of the first iteration starts.

Iteration 1:

1. The Logging phase terminates because the LLB filled up. Recall that the LLB is limited in size, and can only record values from a limited number of load instructions.
2. The processor state (register state and the TLB state) (PS) is checkpointed and the TMR phase is initiated.
3. The TMR phase continues and it has four steps, one for each of the four cores (we call them *TMR steps*).
4. No divergence is seen at the end of the TMR phase. Hence, the next iteration starts.

Iteration 2:

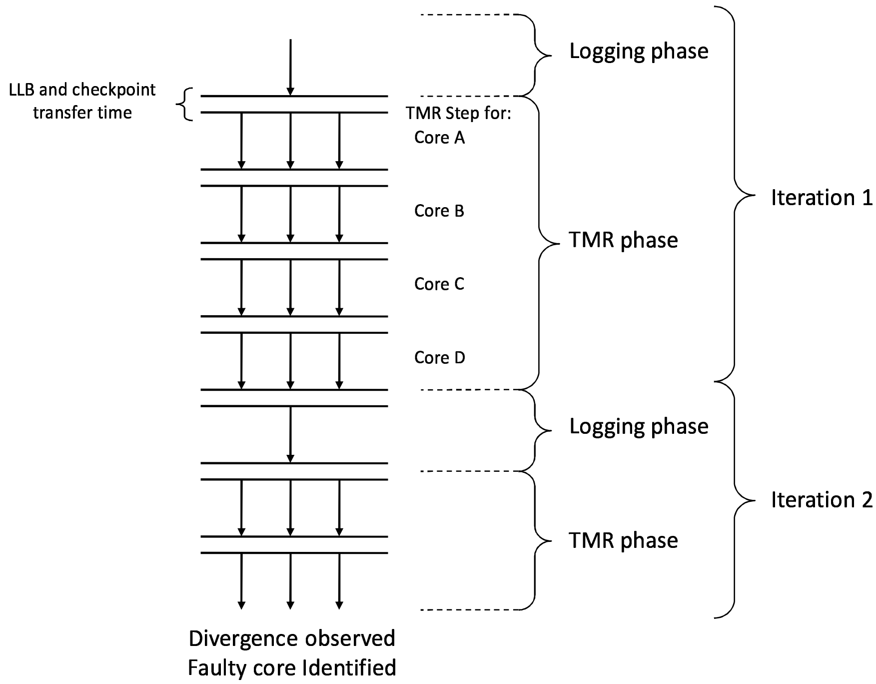


Figure 4.4 The Iterative Multicore Diagnosis Mechanism

1. The Logging phase restarts with the checkpointed processor state (PS) and clear LLBs.
2. The Logging phase terminates due to full LLB and the next TMR phase is initiated.
3. At the end of second TMR step in the TMR phase, a divergence is observed and the faulty core is identified, leading to successful diagnosis.

With the new iterative approach, the diagnosis decision tree (Figure 4.1) changes Figure 4.5. The only change in the diagnosis tree is to check for the termination condition (a symptom in in the Logging phase or if the logging threshold is reached) after the TMR phase. If no divergence is found after the TMR phase and the termination condition is reached after the TMR phase, then we conclude that there is a software bug and let the upper layers of the software handle it. If no termination criterion is satisfied after TMR phase, then the diagnosis continues after the TMR phase as explained above until the termination criterion is satisfied.

4.4.2 Advantages and Disadvantages of Iterative Approach

The advantages of the iterative diagnosis approach are as follows:

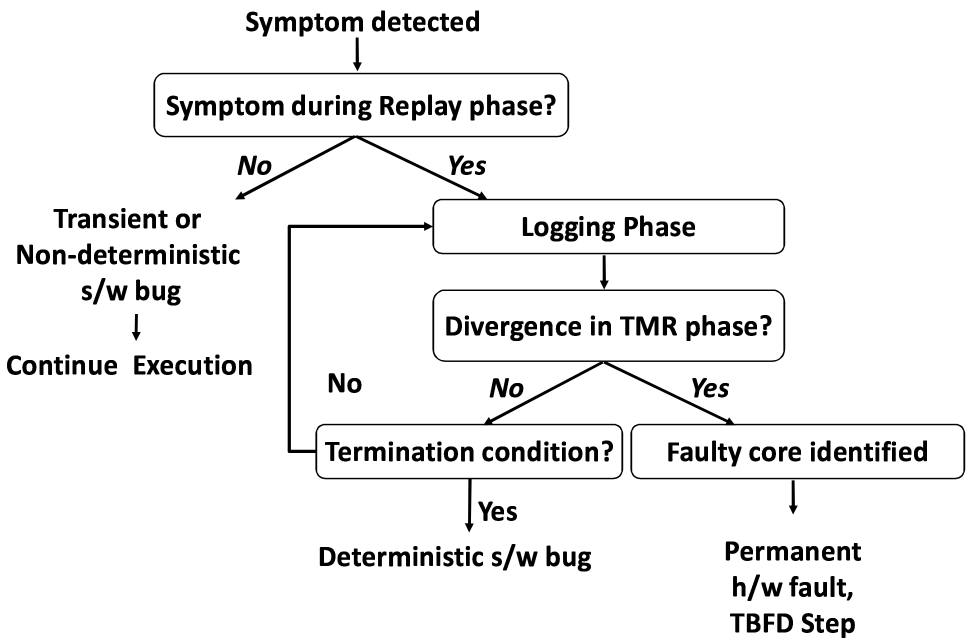


Figure 4.5 The Diagnosis Decision tree for Iterative Multicore Diagnosis Mechanism

- The hardware requirement of the LLB is controllable and can be brought down to a mere few kilobytes (2KB) per core.

- Only the processor state and the TLB state need to be checkpointed. There is no need to checkpoint memory state during Logging and TMR phase. Store instructions are not allowed to write to memory during TMR phase; hence the memory state is untouched from one Logging phase to the other, releasing the requirement of checkpointing memory state.

- The diagnosis latency is shorter for the cases where the fault is activated early in the execution, because we need not wait until the symptom is detected to perform the TMR phase.

This approach has some disadvantages, which are listed below:

- The overhead to transfer the LLB contents and the processor state (registers and TLB contents) across-cores must be incurred at the end of every step in the diagnosis procedure.

- Diagnosability may be somewhat compromised because the microarchitecture state is discarded at the end of each Logging phase. The faults that are active in the microarchitecture state but have not made it to the architecture state will be flushed out. Also, there are cases where the

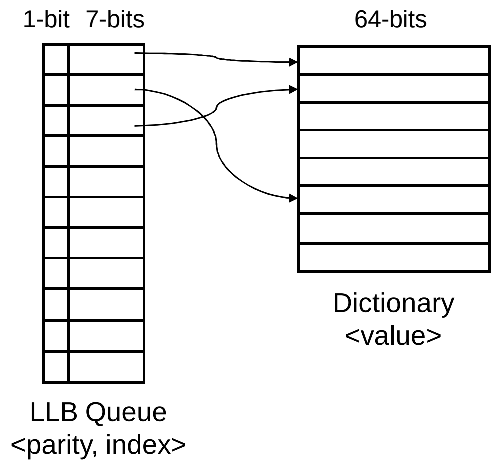


Figure 4.6 Design of Load-Log Buffer (LLB)

fault may not be activated because of the small trace per iteration. Since the microarchitecture is not fully utilized in small instructions traces, there will be fewer fault activations which may lead to no diagnosis (e.g., RAT and Integer register faults). Because of these reasons we want to maximize the instructions in the Logging phase in a given iteration.

4.4.3 LLB Design

The LLB requirements have automatically gone down because we record the load values for limited number of instructions in the iterative approach. At the same time, we want to maximize the number of instructions in Logging phase. The following section explains the design of LLB that can aid in maximizing the number of instructions in Logging phase.

During the Logging phase, the $<$address,value$>$ pairs are logged in the LLB. This information is sufficient to replay the execution of that core in isolation from other cores.

Dictionary based compression: It has been shown that load values exhibit high value locality [41], that is, most of the load values can be captured using a small number of frequently occurring values. Hence, we use a dictionary to record distinct load values, and use a queue that stores the pointers to entries in the dictionary. This method reduces the size of LLB significantly. This method of dictionary based compression was previously used in BugNet [18].

In our approach, we use a 128-entry fully associative table as a *dictionary*. This dictionary stores the load values. Each entry of the dictionary is a 64-bit value. Along with the dictionary, we use a

1024 entry queue (called *LLB queue*) that stores pointers to entries in the dictionary.

Since storing the 64-bit address with each load value can be demanding on area, we propose to store only the parity of the address. We require address only to identify a hit or a miss in LLB during TMR. Instead of storing address A is LLB during Logging phase, we store the parity p_A. In the TMR phase if the address of the load instruction is A, then the same parity is generated and it is considered a hit in LLB. On the other hand, if the load instruction tires to access address B, and if address A and B differ in odd number of bits then the p_A will not be equal to p_B, resulting in a LLB miss. The case when address A and B differ in even number of bits, both the parity bits would match, and it will be considered a hit. Hence, we may lose diagnosability. Since, we are using a single-bit fault model, it is uncommon to have multiple bit flips in the address. There is a trade-off between the area and diagnosability, we choose to reduce area because it is our primary concern.

Hence, we store only the parity (1-bit) instead of storing the entire address (64-bits). We believe that even if there is a hit in the TMR phase (for address, which was supposed to miss), it will corrupt the LLB values for the future load instructions, and we can still diagnose the fault. From our experiments it is evident that storing only the parity is the right trade-off to make.

Therefore, the *LLB queue* entry stores 7-bits for the index into the dictionary, and 1-bit for parity of the address, which totals to 8-bits. The structure of the LLB is shown in Figure 4.6

There are several possible variations of the dictionary approach. We analyze the dictionary based compression mechanism with another variant in Chapter 6. We also explore the effect of varying the LLB size with different dictionary sizes and LLB queue sizes in Chapter 6.

4.5 Firmware

The multicore diagnosis algorithm is implemented in firmware. Since a known fault-free core unavailable, we ensure the correct execution of the diagnosis algorithm in firmware by redundant execution on three cores. A fault detection on a core results in interrupts on three other cores, where the control transfers to the diagnosis firmware. These three cores will execute the instructions in lock-step, which can be emulated, to ensure the correctness of the firmware. Note that the redundant execution is required only in the rare event of fault detection.

CHAPTER 5

EXPERIMENTAL METHODOLOGY

An ideal evaluation of the above discussed techniques to detect and diagnose faults in multiprocessor systems would entail an implementation in the firmware of a real machine and evaluating it under real faults injected in hardware. However, the limited controllability and observability offered by a modern processor would greatly limit our ability to inject and track faults. Potentially, we could use an FPGA-based platform like CrashTest [20] for this purpose. Unfortunately, the limit in size of FPGA would not allow us to directly study a multicore system consisting of superscalar out-of-order processors. Hence, we use simulations to evaluate our scheme. Since it is important to observe the misbehavior of both the application and the OS after hardware faults are injected, the simulator must run fast enough to capture these effects. Hence, we choose a microarchitectural simulator over gate-level simulators. Our methodology is similar to that used for SWAT [11, 10].

5.1 Simulation Environment

We perform microarchitecture-level timing simulation of a chip multiprocessor by using the Wisconsin GEMS timing simulators for the processor and memory [13], in conjunction with the Virtutech Simics full system functional simulator [38]. In this full-system simulation environment, we simulate a 4-core shared memory chip multiprocessor, where each core is a modern out-of-order superscalar processor with a private L1 cache. Table 5.1 gives the parameters of the simulated system.

We run a real operating system (OpenSolaris on Sparc V9 ISA) within this simulated environment and study the behavior of several multithreaded workloads from two benchmark suites under faults. Table 5.2 gives a brief description of the 6 multithreaded workloads we use, along with the input sizes that we study.

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 mult
	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)	64KB
Instruction L1 (private)	64KB
L1 hit latency	1 cycle
L2 (Unified)	4MB
L2 hit latency	6 cycles
L2 miss latency	80 cycles

Table 5.1 Parameters of the simulated processor.

Suite	Workload	Size of input
AlpBench	RayTrace	a teapot scene (2560×2560)
	FaceRec	173 images (130×150)
	Mpeg-Encode	32 HD frames (1920x1080)
	Mpeg-Decode	HD Mpeg video w/ 128 frames
SPLASH-2	lu	1600x1600 matrix
PARSEC	Body-track	4 cameras, 4 frames, 4000 particles, 5 annealing layers

Table 5.2 Workloads and input sizes used in fault injections.

μ**arch structure**	**Fault location**
Instruction decoder	Input latch of one of the decoders
Integer ALU	Output latch of one of the integer ALUs
Register bus	Bus on register file write port
Physical integer register file	An integer physical register
Reorder Buffer (ROB)	Source/destination register number of an instr in ROB entry
Reg Alias Table (RAT)	Logical→ physical mapping of logical register
Address gen unit (AGEN)	Virtual address output

Table 5.3 Fault injection locations.

The GEMS + Simics infrastructure is based on the timing-first approach for simulation [14]. In this approach, an instruction is first executed by the cycle-accurate GEMS timing simulator. When GEMS is ready to retire this instruction, Simics, the functionally accurate simulator, executes the same instruction. The resulting states are compared and in the case that they don't match (which may arise because GEMS does not implement a small subset of infrequently used instructions in the SPARC ISA), the state of the timing simulator is updated from the functional simulator which is assumed to be accurate.

An injected fault in our simulations may, however, also result in this mismatch. For our fault injections, we inject a single fault into the timing simulator's microarchitectural state and propagate the faulty values produced through the system. When a mismatch in the *architectural state* of Simics and GEMS is detected, the state of Simics is synchronized with the faulty state of GEMS. Otherwise, the architectural state of GEMS is updated from Simics, upholding the timing-first paradigm.

5.2 Fault Injection

In this study, we focus on the system-level propagation of permanent faults in the processor core that result in in-field failures. Prior studies attribute the cause of such permanent faults to wear-out and infant mortality due to insufficient burn-in. Such phenomena are expected to become increasingly prevalent with continued CMOS scaling [4]. We model such faults as stuck-at faults (both stuck-at 0 and stuck-at 1) at the latch level. As mentioned, ideally, it is desirable to inject stuck-at faults at the gate level for modeling logic faults. However, doing so will require the use of a gate-level simulator, which is too slow for observing how the fault propagates and manifests as symptoms. Hence, we approximate the stuck-at faults in the logic as stuck-at faults at the latch.

For each experimental run, we inject one fault into a microarchitectural structure within one of the cores while a multithreaded application is running on all cores of the processor. In particular, we inject faults into 7 microarchitectural components of all 4 cores of the simulated processor (Table 5.3). We use the stuck-at fault model to inject faults in various latches in the microarchitecture. For each application, we first pick 5 base injection points (or phases) that are sufficiently spaced apart from each other to capture the different phases of the application's execution. In each phase, for each faulty structure, we pick 5 spatially random injection points (e.g., 5 different physical registers, etc.) and inject both stuck-at-0 and stuck-at-1 faults. This give us a total of 8,400 faults (6 applications $\times$ 5 phases $\times$ 4 cores $\times$ 7 structures $\times$ 5 spatial points $\times$ 2 fault models), a statistically significant sample that yields an error under 0.6% at 95% confidence interval for the fault coverage (defined below) for any structure in our experiments.

We also performed 4,200 transient fault injections (single bit-flip) in the same phases of applications and microarchitectural structures as described above. Since, we have only one fault model, unlike permanent faults, the number of fault injections are fewer.

5.3 Fault Detection

We detect the injected faults using symptoms of anomalous software behavior. Chapter 3 details the symptom detectors we use for our multithreaded workloads. The threshold for the *High OS* detector is set at 50,000 instructions, a higher value than that used in single-threaded applications (previous work used a value of 20k instructions [11]) as synchronization in the multithreaded workloads results in high OS activity on average. The threshold for the *No-Forward-Progress* detector is set at 20,000 instructions for each simulated core. Note that all the cores should simultaneously retire consecutive 20k OS instructions to trigger *No-Forward-Progress*.

These detectors require zero to minimal hardware overhead, with only the *Panic* detector requiring (already existing) support from the OS. Additionally, these detectors present near-zero overhead in fault-free operation, consistent with our motivation of optimizing the fault-free case.

After a fault is injected, we simulate the system for an interval of 20 million instructions (comprised of both application and OS instructions) because 20 million instructions are deemed recoverable. If the fault does not corrupt the architectural state (state of registers and memory) in this interval, the fault is said to be architecturally masked. Faults that are not masked in this interval continue to be simulated in detail for at least 20 million instructions from the time the fault corrupted architectural state (referred to as the fault activation). During this time, they are subject to detection using the above detectors. If the activated faults are not detected in this 20M instruction window, we continue simulation of the application in functional mode (with the Simics functional simulation) until the application completes or a symptom is detected. These cases are not accounted towards fault coverage because it is not deemed recoverable. This allows us to identify application-level masking and silent data corruptions (*SDC*). The fault is not active in this duration of functional simulation, resulting in the permanent fault behaving like an intermittent fault for the detailed simulation window of 20M instructions.

The metrics used to evaluate the efficacy of the symptom detectors are *coverage*, *SDC rate*, and *latency*.

Coverage is the percentage of unmasked (ignoring architectural and application masking) faults detected within 20 million instructions only.

$$Coverage = \frac{Total\, faults\, detected}{Total\, injections - Masked\, faults}$$

where the *Masked faults* are faults masked by either the architecture or the application.

SDC rate is the fraction of the non-masked faults that results in SDCs.

Detection latency computation is a bit more involved as the fault may be detected in a fault-free core that was not injected with the fault. If the fault is detected in the faulty core, the latency is measured in terms of the instructions between the time the architectural state of this core was corrupted and the symptom is detected. However, 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. We measure the latency in terms of the total number of instructions from architecture state mismatch (either the OS or the App) to detection to understand the recoverability of the detected faults.

5.4 Fault Diagnosis

Checkpointing support for rollback: The diagnosis algorithm requires all threads of the execution to be rolled back to a consistent fault-free checkpoint. In our experiments, the system state at the beginning of the fault injection run is checkpointed. We use a scheme similar to SafetyNet [34] where the register state is checkpointed before the execution and the original state of the caches and the memory is logged whenever a cache or memory line is modified for the first time.

Logging phase and LLB: The diagnosis algorithm performs the Logging phase to populate the LLB with load values accessed by the execution. In our microarchitectural simulator, few instructions are incorrectly implemented. For these instructions we record the values from Simics, and use them to replay the execution correctly. To fully understand the feasibility and efficacy of our diagnosis approach, we vary the size of the the LLB and also the LLB structure (discussed in Section 4.4.3). LLB structure and size affect the number of instructions in the Logging phase, which has a direct impact on diagnosis latency. Chapter 6 discusses these trade-offs in detail.

TMR phase: The TMR phase replays the execution on three cores and collects the information to be compared at the end of the TMR step. As mentioned in Chapter 4.4.2, we do not allow store instructions to write to memory. We capture this effect in our simulator by recording the value before retire, and writing the same value after the store instruction is retired (we do this because we cannot control the way simics retires instructions). Since some instructions are not implemented in GEMS, we cannot retrieve the value of the store instruction before it retire. In such cases, to keep the memory state unchanged, we squash the store instruction, and proceed with the next TMR step. Diagnosability may be compromised because we cannot continue execution in the current TMR step.

Diagnosis latency: Our diagnosis algorithm first logs the execution in LLB during the Logging phase, transfers the LLB to other cores, and then uses it during the TMR phase to deterministically replay the execution. To compute the diagnosis latency we need to account for the transfer time along with the latency of each phase. In our simulations, we do not accurately model the transfer of the processor checkpoint and the LLB across cores. In order to obtain realistic diagnosis latency, we add the estimated transfer time at the end of each iteration.

We estimate the time to transfer the LLB and processor checkpoint as follows. First, we compute the size of the checkpoint and the LLB in terms of cache lines. Assuming a 64 byte cache line, the processor checkpoint consists of 427 registers[1], each 64 bits wide, that corresponds to 54 cache lines. The LLB, a 2KB structure, corresponds to 32 cache lines.

At the end of Logging phase, the firmware flushes the LLB and the processor checkpoint to memory. For a representative processor, we assume an L2 cache with 8 MSHRs and memory access latency of 200 cycles. Thus the 86 cache lines corresponds to 2150 cycles of transfer time (since 8 outstanding misses can be handled). For a 4 core system, we conservatively estimate this time to be 8600 cycles (2150 $\times$ 4).

In the TMR phase, the LLB and the processor checkpoint need to be loaded from memory. We estimate this transfer time to be 2150 cycles for each core, as per our previous calculation. In TMR phase, only 3 cores operate at any given time, resulting in a cumulative 6450 cycles to load LLB and processor checkpoint. In one iteration LLB and processor state are flushed to memory after Logging phase, and are loaded from memory for each of the 4 TMR steps in the TMR phase. Hence we add (8600 + 4$\times$6450) 34,400 cycles at the end of each iteration to account for the overhead due to the transfer of LLB and processor checkpoint.

The above mentioned overhead is conservative because of the following two reasons. First, the execution from the next TMR step can hide most of the memory latency. Second, most of the memory accesses will hit in L2 (while estimating the overhead, we assumed every access to be a miss in L2) as contents of memory are not modified in the TMR phase.

We believe that comparing the executions in the TMR phase can be performed with negligible overhead by using hardware signatures, as explained in Section 4.3.3.

Page faults: We replay the page faults deterministically during diagnosis. Since, a page fault is initiated by a TLB miss, and TLB is software managed in our system, the page fault would be initiated again during TMR phase, if there was a page fault in the Logging phase. Since, we do not allow the store instructions to commit, the illusion of a page fault can be created on all the cores running in parallel. Replaying page faults is not necessary for our technique.

Disabling Interrupts: In our work, we disabled asynchronous interrupts during diagnosis. While most interrupts in SPARC V9 are disabled, cross calls between the processors in a multi-processor setting are always serviced and cannot be disabled (at interrupt level 15). Hence, when we see such

[1](4 sets of globals $\times$ 8) + (8 windows $\times$ 32 integer registers) + (32 floating point registers) + (107 control registers)

a cross-call, we abort and restart the current step in the diagnosis (logging, or any intermediate TMR step) to ensure determinism in the replay. A disadvantage with this approach is that if the fault is activated only by the interrupt handler, it would not be diagnosed as we never execute that piece of code in our diagnosis. However, we could extend our mechanism to handle such cases by logging interrupts, by a mechanism similar to that proposed in previous work [40].

CHAPTER 6

RESULTS

6.1 Multicore Fault Detection

SWAT [11] has shown that using software symptoms gives significantly high coverage (over 95%) for faults injected in various microarchitectural units in a uniprocessor system running a single threaded workload [11]. As discussed previously, we extend the SWAT symptoms developed for single core processors to the multi-core counterparts and study the effectiveness of these detectors for faults in multi-core systems by measuring the coverage of the detectors and the latency to detect the injected faults.

6.1.1 Coverage

Figure 6.1 gives the coverage of the symptom detectors for the permanent faults injected across seven different microarchitecture structures in the processor core. For faults injected in each microarchitectural unit (shown on the X-axis), the figure shows the percentage of faults that are Masked (by both the architecture and application), detected by a symptom from the faulty core (*Detect-Faulty*), detected by a symptom in a fault-free core (*Detect-Fault-Free*), detected by a symptom in functional mode (*Symp>20M*, not counted towards coverage), and Silent Data Corruptions (*SDC*). The numbers on top of each bar show the coverage of the symptom detectors for faults in that unit.

The software-level symptoms perform exceptionally well to detect underlying permanent hardware faults in multithreaded workloads running on multicore systems. Figure 6.1 shows that these symptom detectors show a very high coverage, an average of 98.8% across all structures, with a low SDC rate of 0.84%.

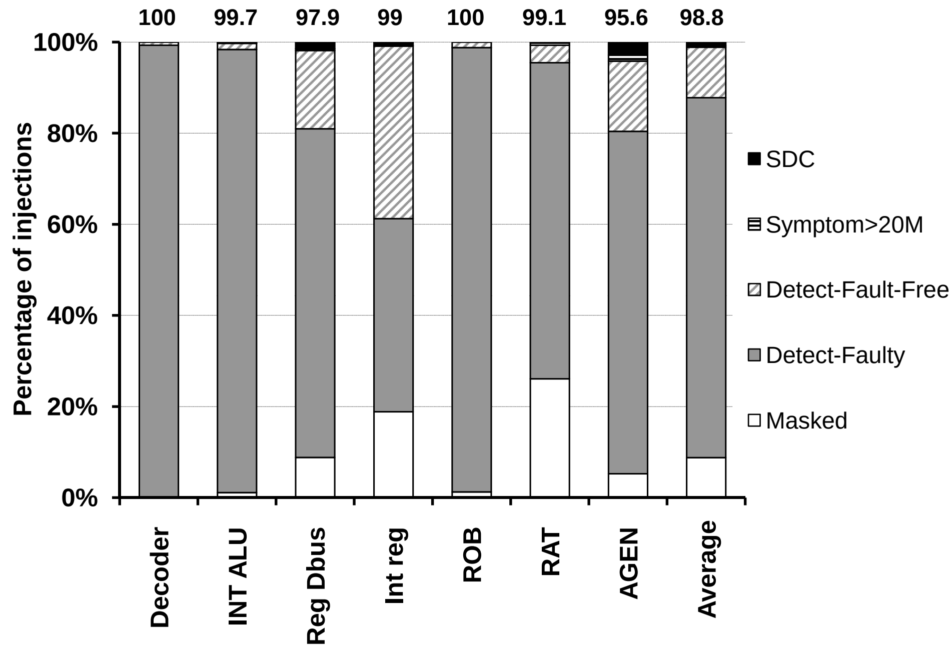


Figure 6.1 Coverage of the symptom-based detectors in multicore systems running multi-threaded workloads.

Figure 6.1 also shows that in about 12.3% of the detections, the symptom is detected on a core that is fault-free (we consider *Panic* and *No-Forward-Progress* as symptoms detected in a fault-free because these affect the execution in fault free cores). Since the simulated system is a shared memory machine, these faults propagate through memory into a fault-free core on which they affect the executing software. It is this fault propagation that complicates the diagnosis of the fault-free core, as previously discussed (Chapter 4).

Figure 6.2 further categorizes the faults detected in the faulty and fault-free cores. For each faulty structure, the figure categorizes the detected faults as detected on *Faulty* and on *Fault-free* cores. For each such category, the bars show the distribution of the various symptoms that triggered detection. For fatal traps, we distinguish between cases where the trap was thrown by the application code (FatalTrap-App) and by the OS code (FatalTrap-OS). Similarly, we distinguish between hangs in the application (Hang-App) and the OS (Hang-OS). The height of each bar is the percentage of total faults injected in that structure that triggered a detection in that category.

Similar to the fault injections in single-threaded workloads [11], we see that symptoms from the OS (*FatalTrap-OS*, *High-OS*, *Hang-OS*, *Panic*, *No-Forward-Progress*) account for a large fraction of

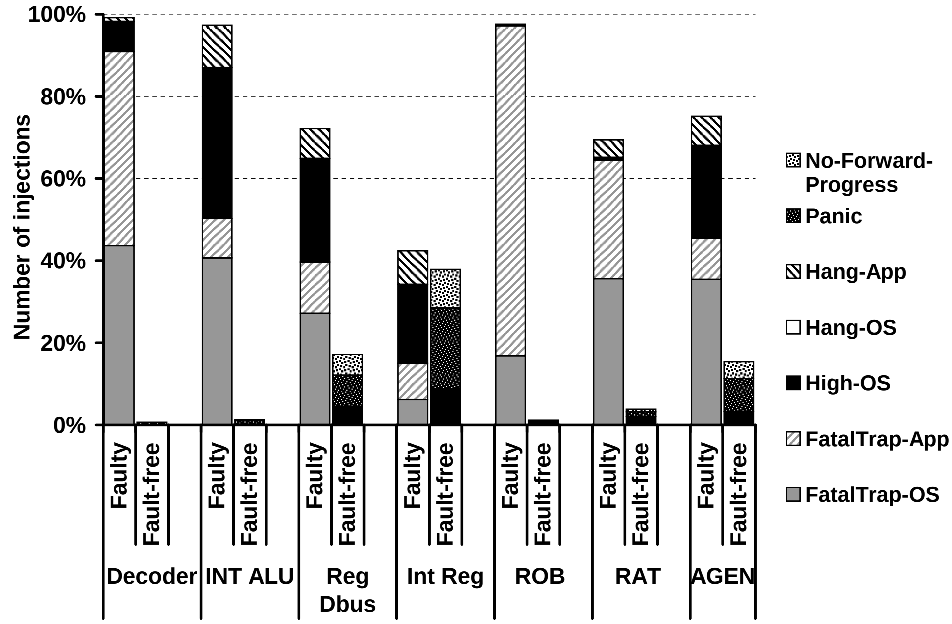
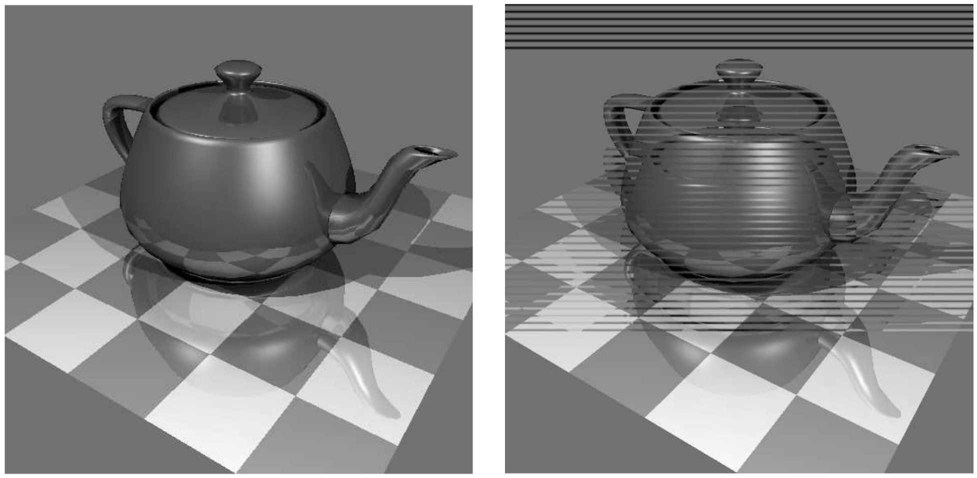


Figure 6.2 Contribution from each detector in detecting faults in multicore systems.

the detections on the faulty core (62.6% of detected faults, on average). Although the fault is injected while the application is running, the fault quickly results in invoking the OS (typically through a TLB miss). The OS, being highly control intensive, activates the fault further causing anomalous behavior that results in a symptom. Hence, a large fraction of faults detected in the faulty core are detected in the OS.

SDC: Silent Data Corruptions are those cases where no symptom was thrown and the output generated by the faulty execution did not match the golden output. 0.85% of the unmasked faults resulted in SDC. We use simple file comparison to determine the silent data corruption. For several applications, simple comparison (*diff*) of the output may be too conservative in identifying the SDC. There has been a study in the context of transient faults showing that the 45% of the faults that corrupt architecture state are correct at application level for multimedia and AI applications [12]. This study uses signal-to-noise ratio to identify acceptable output.

We have also observed a similar trend in the multimedia applications, though we use only simple file comparison for reporting results. There are several cases in RayTrace, MPEG-decoder and MPEG-encoder where there was no human noticeable differences even though the output is different from the correct output. On the other hand, we also observe that few outputs had noticeable



(a) Golden output

(b) Silent Data Corruption fault from RAT

Figure 6.3 RayTrace outputs: (a) Output when the program was run on native machine with no fault injection. (b) SDC case from the fault injection experiment when the fault in the register alias table.

differences. Figure 6.3 shows one such example (the only case from RayTrace that had noticeable corruption).

We did not evaluate the correctness of the output generated by the faulty runs. This is one of our future directions.

6.1.2 Latency

In addition to fault coverage, an important parameter for any detection scheme is the latency at which the faults are detected. The detection latency has direct repercussions on the recoverability of the system; if the latency is too long, the checkpoint interval will need to be lengthened accordingly. Long checkpoint intervals affect both the restart time on recovery and input/output commit latencies. If not properly handled, outputs could be committed erroneously, making the system state unrecoverable. Therefore, we use detection latency as a metric to evaluate the proposed symptom detectors.

Figure 6.4 shows the latency, from the first architecture state corruption (of either the OS or the application) to detection. For each microarchitecture structure, the bar is divided into different latency stacks (from 1k to 20M), each representing the percentage of the detected faults with latency lower than the specific detection latencies.

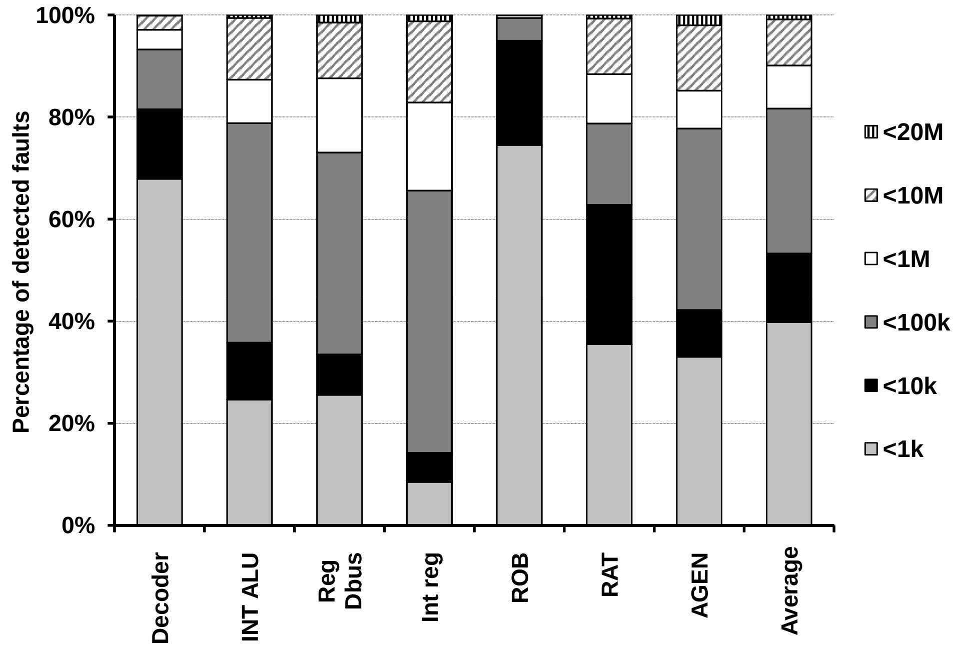


Figure 6.4 Latency to fault detection, in instructions, from first architecture state mismatch to detection.

From the figure we see that, 90.13% of the detected faults have detection latency of under a million instructions, 9% have latency between 1M and 10M instructions, 0.86% are detected after more than 10M instructions. Hardware techniques such as ReVive [21] and SafetyNet [34, 40] can comfortably tolerate checkpoint intervals of hundreds of milliseconds. However, the output will have to be buffered and delayed for this interval as discussed in [16]. The multithreaded workloads studied here are not I/O intensive. For other workloads, it will be important to investigate the impact of buffering the I/O events and is the subject of our future work.

6.1.3 Transient Faults

From the transient fault experiments, we found that 85.4% of the faults were architecturally masked. Of the remaining faults, 40.7% were detected within 20 million instruction window (92% of which are detected on faulty-core and 8% on a good core). The remaining faults were run to the completion, and we found that 64% of these cases were masked at application level, and 7% were eventually detected as symptoms. The remaining resulted in silent data corruptions. This makes the overall masking rate to 91%, detection rate (for unmasked faults) to 41% and the SDC rate to 2.5%. Most of the SDCs, 85%, were from MPEG-decoder alone, most of which affected only one frame. Since these applications are inherently fault tolerant [12], we believe these silent data corruptions are acceptable. These results are consistent with previous studies [11, 39].

Overall, we see that using symptom detectors that watch for anomalous software behavior is highly effective in detecting hardware faults in multicore systems. In addition, the latencies to detection make a large fraction of the detected faults amenable to hardware recovery.

6.2 Multicore Fault Diagnosis

After the detection of a symptom, diagnosis is invoked to identify the faulty core. Specifically, for the 12.3% of the detected faults that resulted in symptoms from fault-free cores (from Figure 6.1), it is important to see whether the diagnosis algorithm successfully diagnoses the faulty core.

In this section, we evaluate the diagnosability, diagnosis latency, and the LLB size requirement of our method.

6.2.1 Diagnosability

Figure 6.5 compiles the diagnosis outcomes of the detected faults across different microarchitectural structures. *CorrectlyDiagnosed* shows the cases where the core injected with the permanent fault is correctly diagnosed. *Undiagnosed* shows the cases where the fault is not diagnosed. The numbers on top of each bar show the percentage of detected faults that are diagnosed correctly.

We confirmed that 100% of the faults that were detected on fault-free cores (12.3% of the detected cases from Figure 6.1) were successfully diagnosed.

As mentioned in Section 4.3.3, during TMR phase we compare branch targets and store values and use LLB Hit/Miss information to identify divergence. From our experiments, we found that comparing the entire execution (destination registers of every retiring instruction) increases the coverage by 0.25%. Since, the benefit is small, we do not compare the entire trace, although, we believe, the overhead in comparing the destination registers of all the retiring instructions will not be large. Section 4.3.3 gives details about the ways in which this comparison can be performed with low overhead.

Figure 6.5 shows that 4.35% of the faults do not lead to successful diagnosis after replaying the execution for 20 million instructions. We found that most of these cases (79% of the undiagnosed)

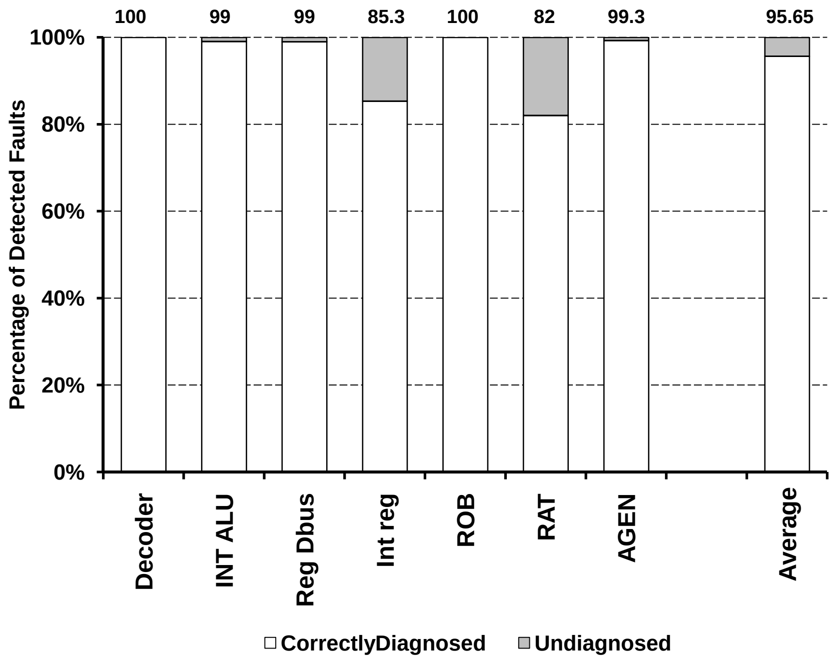


Figure 6.5 Diagnosability of faults in multicore systems.

did not activate the fault during the diagnosis. This leads us to the conclusion that the fault is not activated by the short instruction traces in our iterative mechanism because the microarchitecture is not fully utilized.

6.2.2 Diagnosis Latency

Even though diagnosis is invoked rarely and can tolerate higher performance overheads, shorter diagnosis latency is preferable in order to be able to bring the fault-free cores back online without human perceivable delay.

We compute the diagnosis latency from the beginning of the diagnosis phase until the end of the diagnosis algorithm. This includes multiple logging and TMR phases. The computation of the diagnosis latency is explained in Section 5.4.

Figure 6.6 shows the total diagnosis latencies of our diagnosis mechanism. We observe that 89.6% of diagnosed cases had latency within 100,000 cycles, and 95.7% of the cases were diagnosed within 1 million cycles. The number of the top of each bar shows the percentage of diagnose cases with latency less than 1 million cycles. This shows that our diagnosis mechanism is capable in diagnosing the faults with very low latencies.

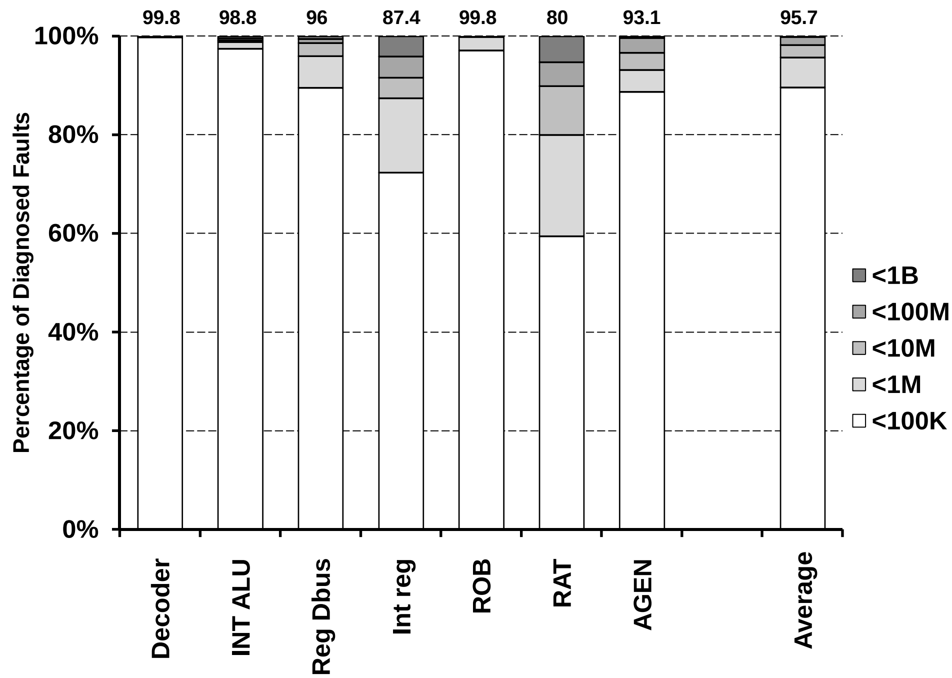


Figure 6.6 Diagnosis latency of faults in multicore systems (in cycles).

From Figure 6.6, we observe that the diagnosis latency for the undiagnosed cases can be large (>100 million cycles). Since our diagnosis algorithm identifies undiagnosed cases as software bugs, it will be undesirable to use this technique in the development phase, when the software bugs are more frequent. If the execution overhead during application development phase is noticeable, one can choose to turn-off the diagnosis mechanism. Since the symptom detection is a rare event (either due to software bug or hardware fault), we believe the higher overhead for diagnosing software is acceptable.

Since we have few cases (8% of the detected faults) where the diagnosis latency is larger than 1 million cycles, we investigate the possible reasons for the execution overhead in our iterative approach (Section 4.4.1), which are the following: (1) Time to transfer LLB and processor checkpoint across cores after each step during diagnosis. (2) Squashes introduced in the TMR phase while ensuring deterministic replay (Section 4.3.1).

In order to understand the overhead due to the first two reasons, we plot the number of iterations (number of Logging phases) in our iterative approach. Figure 6.7 shows the number of iterations for all the cases that were subjected to diagnosis. In each iteration, the pipeline is squashed 5 times

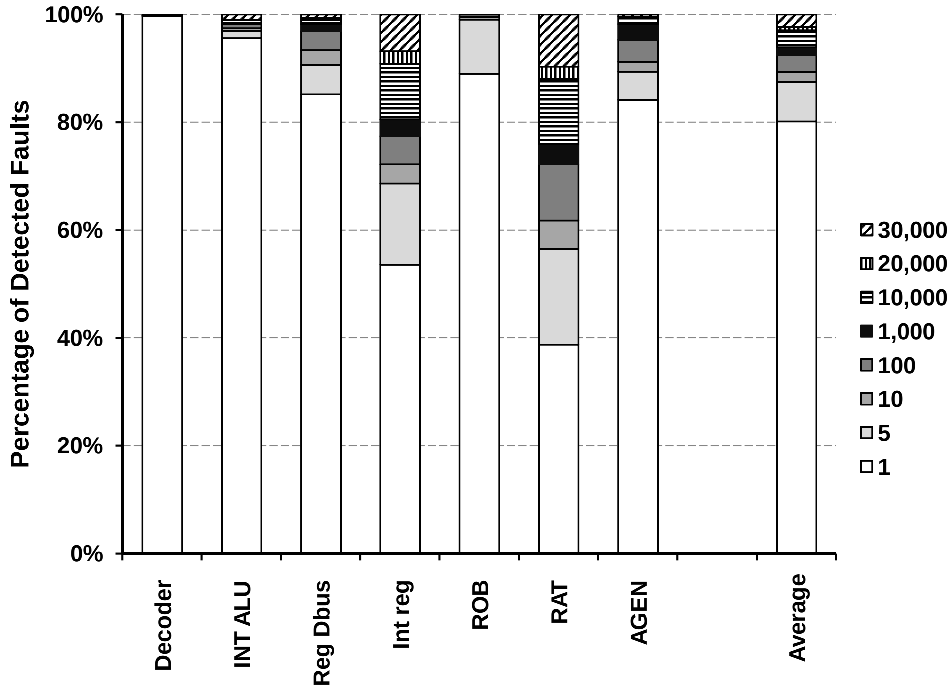


Figure 6.7 Number of iterations (Logging phases) before diagnosis.

and 5 cross-core transfers take place (1 after Logging phase, 1 after each of the 4 TMR steps during TMR phase). We observe that the 6% of the detected faults require at least 1000 iterations in our diagnosis approach, and 3% require at least 10,000 iiterations. Notably, the overhead is high for the undiagnosed cases, which require a large number of iterations (>10,000). Since, the number of iterations depends on the LLB size, we study the effect of different LLB sizes and structures in the following section.

6.2.3 LLB Size and Structure

As mentioned in Section 4.4.3, we want to maximize the number of instructions in the Logging phase of each iteration. The above results also state that increasing the number of instructions will reduce the overhead due to cross core transfers, which in-turn reduces diagnosis latency.

The number of instructions in each iteration is limited by the capacity of the LLB. Once the LLB is full, subsequent instructions can not be logged. To understand the limitation due to LLB, we perform diagnosis with two different LLB structures (explained in Figure 6.8). LLB structure A is similar to one that is used in BugNet. We observed that the many of the load values differ in least

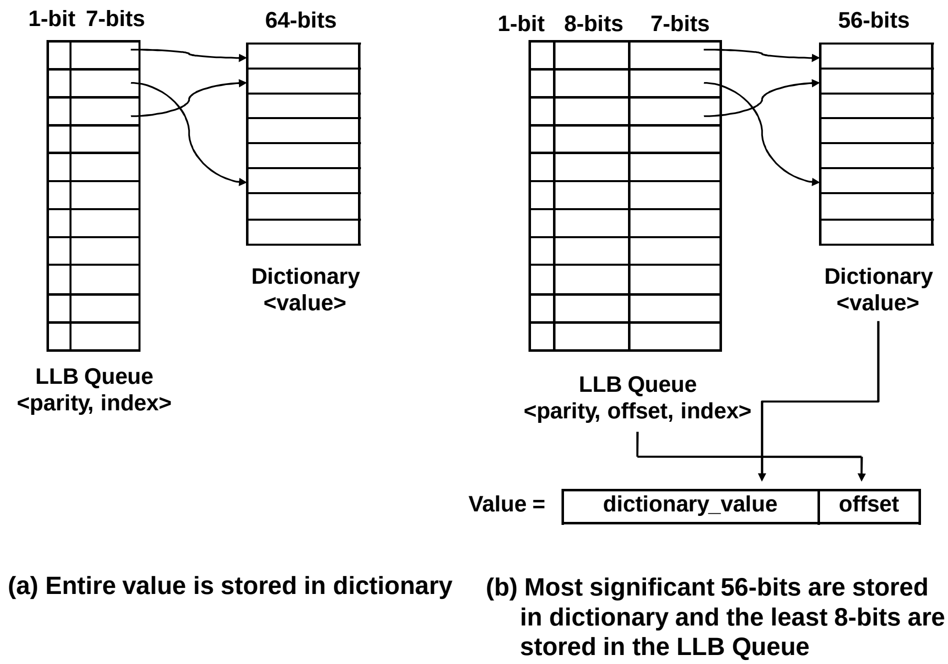


Figure 6.8 LLB structures. (a) LLB structure A. (b) LLB structure B.

significant 8-bits. Hence, we evaluate structure B and compare it with structure A. To identify the most appropriate LLB structure, we measure the number of instructions in the Logging phase in one iteration during diagnosis. As the locality in the load values is primarily based on the applications, we categorize these results based on applications. Each bar in Figure 6.9, and 6.10 is computed as follows: (1) Number of instructions in the Logging phase of each iteration is averaged across 100 iterations of the diagnosis algorithm. (2) We compute the average number of instructions in the Logging phase at 5 different phases per application. Since, we are studying the cases that lead to long diagnosis latencies, specially the undiagnosed cases where the fault is rarely activated, we obtained these graphs from the fault-free cases.

Figure 6.9 shows the number of instructions in the logging phase for the LLB structures A and B respectively. In Figure 6.9, the dictionary has 128 entries and the LLB queue has 1024 entries. Hence, the LLB size is fixed to 2KB and 2.87KB respectively, for LLB structures A and B.

From Figure 6.9, we see that LLB structure B can record more number of instructions than LLB structure A, but the benefit is low for the additional hardware cost. Hence, we choose LLB structure A.

Since the number of instructions in Logging phase also depends on the size of LLB, we study the

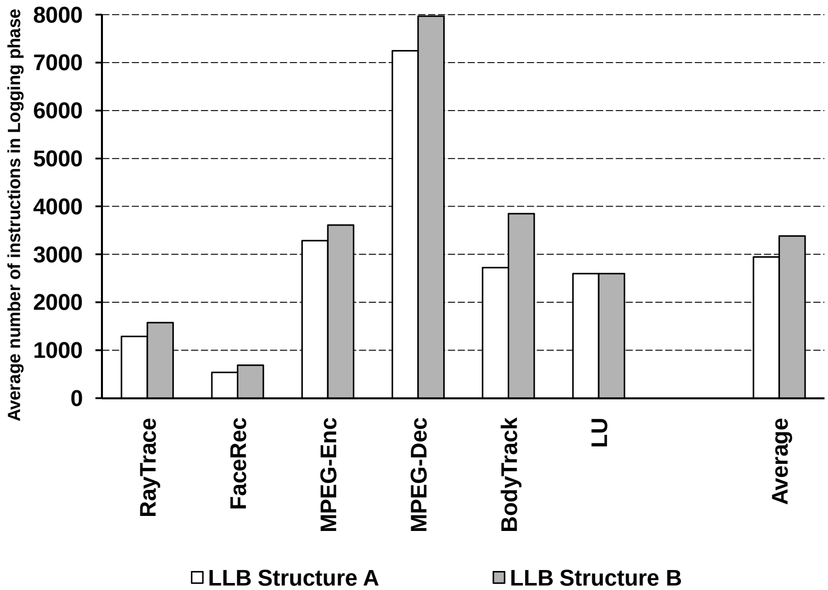
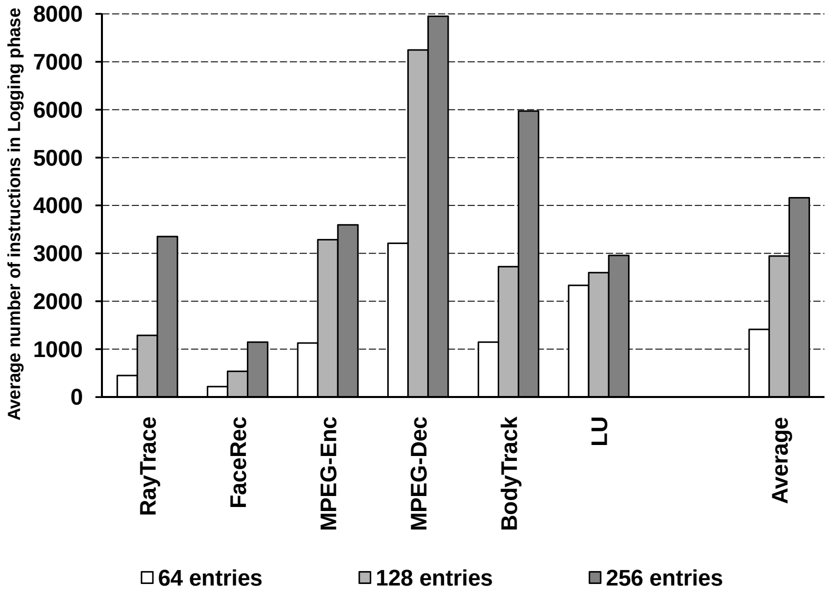


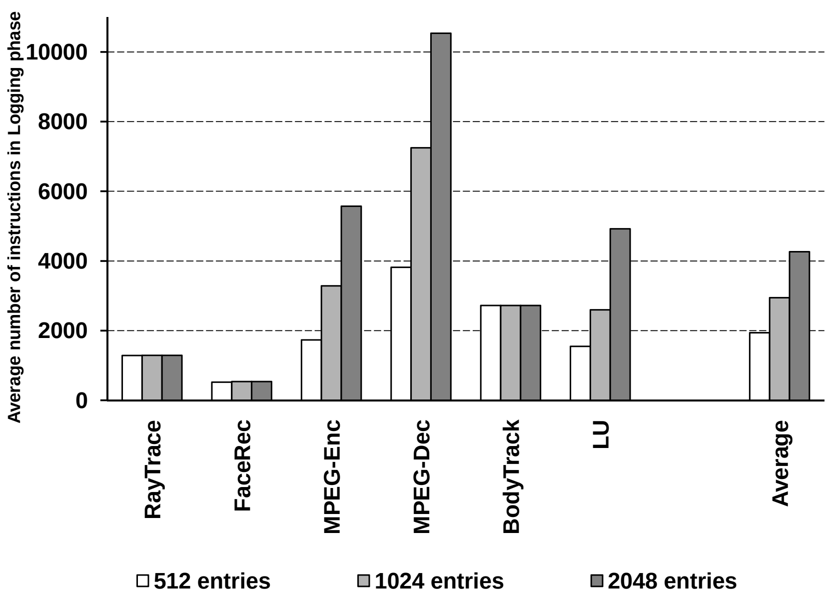
Figure 6.9 Average Number of instructions in the Logging phases due to different LLB structures. Size of LLB Structure A is 2KB and structure B is 2.87KB

effect of the varying the LLB size on the number of instructions in the Logging phase. Figure 6.10(a) shows the number of instructions in the Logging phase when the dictionary size is varied from 64 to 256 entries. In these results the size of the LLB queue is kept constant with 1024 entries. Figure 6.10(b) shows the results when the LLB queue size is varied from 512 to 2048 entries. In these results, the dictionary size is kept constant with 128 entries.

From Figure 6.10 we see that the number of instructions in the Logging phase is bound by the dictionary size for some applications and by the LLB queue size for others. We observe (Figure 6.10(b)) that the number of instructions in the Logging phase for RayTrace, FaceRec and BodyTrack did not increase when the LLB queue size is increased from 512 to 2048 entries. On the other hand, MPEG-Enc, MPEG-Dec and LU are limited by the dictionary size. This is evident from Figures 6.10(a). Since the dictionary is a fully-associative structure, we have additional penalty for having larger dictionary size. Therefore, we decided to use the LLB with 1024 entries in the LLB queue and 128 entries in the dictionary.



(a) Effect of varying dictionary size on the number of instructions in Logging phase



(b) Effect of varying LLB queue size on the number of instructions in logging paste

Figure 6.10 Average number of instructions in the Logging phase when the LLB size is varied. (a) Different dictionary size but same LLB queue size (1024 entries). Size of LLB with 512 dictionary entries is 1.5KB, with 1024 entries is 2KB, and with 2048 entries is 3KB. (b) Different LLB queue size but same dictionary size (128 entries). Size of LLB with 64 LLB queue entries is 1.5KB, with 128 entries is 2KB, and with 256 entries is 3KB.

μarch structure	Percentage of load instructions squashed
Decoder	0.34
INT ALU	0.34
Reg Dbus	2.68
Int reg	2.45
ROB	1.70
RAT	2.34
AGEN	2.07
Average	2.25

Table 6.1 Percentage of load instructions squashed during diagnosis.

6.2.4 Overhead for Ensuring Deterministic Replay

We evaluate the overhead due to the squashes introduced by the load instructions in the TMR phase while ensuring deterministic replay (as explained in Section 4.3.1). Table 6.1 lists the percentage of the load instructions that were squashed in the TMR phase. On average, 2.25% of the load instructions were squashed, which is fairly low.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

Low-cost hardware reliability solutions that do not rely on excessive redundancy are becoming increasingly important. Recent advances in solutions that use anomalous software behavior (or symptoms) to detect hardware faults appear very promising. These solutions incur almost no overhead during the common fault-free case, relegating most of the overhead to the rare case of when a fault is detected. However, the detection and diagnosis techniques based on this approach have previously only been studied for single threaded applications running on a single core. For these methods to become practically useful, they must be applied to multicore systems with multithreaded applications.

To the best of our knowledge, this is the first work that tackles both detection and diagnosis for faults in multithreaded workloads running on multicore systems, without the use of excessive redundancy during fault-free operation. It addresses the challenge of diagnosing the faulty core in the multi-core system, as the assumption that the symptom-causing core is faulty breaks when the fault propagates from one core to another.

We extended the symptoms developed for single core systems and evaluated them in our simulated system. Our results are promising, and show that the detectors have a very high coverage of 98.8% at a low SDC rate of 0.84%.

Further, 95.7% of the detected faults subjected to diagnosis were successfully diagnosed. Although this diagnosis is rarely invoked in the event of a fault, we optimize our diagnosis algorithm to lower the diagnosis latency. On an average, the diagnosis algorithm successfully diagnosed 96% of the cases within one million cycles (which corresponds to 1ms on 1GHz systems). We also optimized our technique for minimal hardware overhead and reduce the overhead to a mere 2KB per core.

7.2 Limitations and Future Work

- *Fault detection coverage*: We show that the SWAT approach to detect hardware faults through software-level symptoms provides high coverage of 98.8% in multicore systems. This coverage can further be improved by using iSWAT [26], and assertion-based microarchitecture-level checkers [23]. iSWAT proposed the use of likely program invariants as a symptom, and showed that the silent data corruptions can be reduced significantly thereby increasing the coverage. Our detectors can also be complemented with hardware checkers [23, 24] for higher coverage.

  We believe that formally deriving hardware checkers that can detect hardware faults that escape our symptom detectors is a promising direction for future research.

- *Fault detection latency*: The approach of SWAT is to watch for software anomalies as symptoms of a hardware fault. Since some faults can have long latency to detection, recoverability can become an issue. Although we show that 98.8% of the unmasked faults are detected within 20 million instructions (Section 6.1.2), there are few cases (0.4% of unmasked faults) that are detected after 20 million instructions and may require additional support for recovery. Compiler assisted techniques can be used to reduce the detection latency, similar to one used in iSWAT. Using hardware checkers can also greatly reduce detection latency and is one of our future work.

- *Diagnosability*: The iterative diagnosis approach performs diagnosis through repeated replay of execution of small traces. These small traces may not fully utilize the microarchitecture. Hence the fault may not be activated by those small traces resulting in no diagnosis. We provide a tradeoff between the area and diagnosability in Section 6.2.3, and show a way in which the number of instructions in each trace can be maximized. We believe that techniques similar to one explained in Section 4.4.3 can be enhanced to further increase the number of instructions in each trace.

- *Reliable firmware execution*: Our detection, diagnosis, and recovery mechanisms are controlled by firmware. The firmware provides sufficient visibility of the software and hardware to implement our low-cost technique. We rely on reliable firmware execution. In Section 4.5 we discussed a solution for reliable firmware execution. We believe this solution can further be improved to lower the execution overhead, which is one of our future directions.

- *Off-core faults*: With the advent of multicore era, hardware faults in off-core components are becoming more important. This thesis provides a low-cost symptom-based fault detection mechanism that is evaluated in the context of in-core faults. The applicability of this mechanism for off-core faults is unclear. After detection, there is a need to identify the fault component, and to the best of our knowledge, there is no work that distinguishes off-core faults from in-core faults. Hence diagnosing off-core faults is still a challenge. We believe proving the applicability of SWAT and designing a diagnosis mechanism for off-core are promising directions for future research.

REFERENCES

[1] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture*, 1998.

[2] D. Bernick et al. NonStop Advanced Architecture. In *International Conference on Dependable Systems and Networks*, 2005.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7):422–426, 1970.

[4] S. Borkar. Microarchitecture and Design Challenges for Gigascale Integration. In *International Symposium on Microarchitecture*, 2005. Keynote Address.

[5] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *International Symposium on Computer Architecture*, pages 227–238, 2006.

[6] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-Based On-Line Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *International Symposium on Microarchitecture*, 2007.

[7] M. Dimitrov and H. Zhou. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *International Conference on Parallel Archtectures and Compilation Techniques*, 2007.

[8] 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.

[9] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *International Symposium on Computer Architecture*, pages 172–183, 2005.

[10] 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 *International Conference on Dependable Systems and Networks*, 2008.

[11] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[12] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *International Symposium on High Performance Computer Architecture*, pages 181–192, 2007.

[13] M. Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

[14] C. Mauer, M. Hill, and D. Wood. Full-System Timing-First Simulation. pages 108–116, 2002.

[15] A. Meixner, M. E. Bauer, and D. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *International Symposium on Microarchitecture*, 2007.

[16] J. Nakano et al. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *International Symposium on High Performance Computer Architecture*, 2006.

[17] N. Nakka et al. An Architectural Framework for Detecting Process Hangs/Crashes. In *European Dependable Computing Conference (EDCC)*, 2005.

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

[19] K. Pattabiraman et al. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *European Dependable Computing Conference*, 2006.

[20] A. Pellegrini et al. Crashtest: A fast high-fidelity fpga-based resiliency analysis framework. In *IEEE International Conference on Computer Design*, September 2008.

[21] M. Prvulovic et al. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, 2002.

[22] P. Racunas et al. Perturbation-based Fault Screening. In *International Symposium on High Performance Computer Architecture*, 2007.

[23] V. Reddy, A. S. Al-Zawawi, and E. Rotenberg. Assertion-Based Microarchitecture Design for Improved Fault Tolerance. In *International Conference on Computer Design* , 2006.

[24] V. K. Reddy and E. Rotenberg. Coverage of a microarchitecture-level fault check regimen in a superscalar processor. In *International Conference on Dependable Systems and Networks*, pages 1–10, 2008.

[25] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *International Symposium on Fault-Tolerant Computing*, 1999.

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

[27] E. Schuchman and T. Vijaykumar. Blackjack: Hard error detection with redundant threads on smt. *Intl. Conf. on Dependable Systems and Networks*, pages 327–337, 2007.

[28] Design Panel, SELSE II - Reverie, 2006. `http://www.selse2.selse.org/recap.pdf`.

[29] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S. W. Keckler. Scalable hardware memory disambiguation for high-ilp processors. *International Symposium on Microarchitecture*, 24(6):118–127, 2004.

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

[31] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *International Symposium on Microarchitecture*, 2006.

[32] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *Proc. of 11th Intl. Conf. on Architectural support for programming languages and operating systems*, 2004.

[33] J. C. Smolens, J. Kim, J. C. Hoe, and B. Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarch.s. In *International Symposium on Microarchitecture*, 2004.

[34] D. Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *International Symposium on Computer Architecture*, 2002.

[35] 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.

[36] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, 2000.

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

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

[39] 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.

[40] M. Xu, R. Bodik, and M. Hill. A “Flight Data Recorder” for Enabling Full-system Multiprocessor Deterministic Replay. In *International Symposium on Computer Architecture*, 2003.

[41] J. Yang and R. Gupta. Energy efficient frequent value data cache design. *International Symposium on Microarchitecture*, 0:197, 2002.

[42] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *International Symposium on High Performance Computer Architecture*, pages 261–272, 2007.