CASPAR: Hardware Patching for Multi-core Processors

Ilya Wagner
iwagner@umich.edu

Valeria Bertacco
valeria@umich.edu

University of Michigan, Ann Arbor, MI 48109

ABSTRACT

Ensuring correctness of execution of complex multi-core processor systems deployed in the field remains to this day an extremely challenging task. The major part of this effort is concentrated on design verification, where different pre- and post-silicon techniques are used to guarantee that devices behave exactly as stated in the specification. Unfortunately, the performance of even state-of-the-art validation tools lags behind the growing complexity of multi-core designs. Therefore, subtle bugs still slip into released components, causing incorrect computational results, or even compromising the security of the end-user systems.

In this work we present Caspar – an approach for in-the-field patching of the memory subsystem hardware in multi-core chips. Caspar relies on a checkpointing system, which periodically logs the state of the chip, and a novel error detection and recovery scheme, which uses a simplified mode of operation to bypass cache coherence and consistency errors. The implementation of Caspar employs hardware detectors: on-die programmable circuits to identify system's configurations that may lead to bugs, and to trigger recovery and bypass. Our experimental results show that Caspar can be used effectively to detect and bypass a variety of memory subsystem bugs, with as little as 2% performance impact and 6% area overhead during bug-free operation.

1. INTRODUCTION

Correctness of execution is one of the most crucial characteristics of any digital design and, in particular, microprocessors, which permeate our modern surroundings. Unfortunately, processor manufacturers cannot provide guarantees for the correctness of these products due to their extreme complexity. Despite the best efforts in pre- and post-silicon validation, subtle corner case interactions between on-chip components are often overlooked and slip through the verification process. Once systems are deployed in the field, these erroneous behaviors may lead to effects ranging from relatively innocuous to devastating. Examples of these critical issues include the F00F bug [1] and a recent bug in AMD's Phenom chips [2]. The last one is particularly interesting because it occurred in a multi-core design – an increasingly common architecture today. Multi-core chips offer significant performance advantages, but this benefit comes at the cost of a much more involved verification effort, since the communication among the system's cores must also be validated. With the number of cores growing to 64 [5], 80 [14] and beyond, the complete verification of the communication protocol becomes prohibitively complex, allowing bugs to seep into released products.

In the majority of modern multi-core processors, the communication protocols are implemented through shared memory and, therefore, require verification of *memory coherence* and *consistency*. Memory (or cache) coherence guarantees that individual cache lines are properly handled, so that any read access always returns the value from the last write. Unfortunately, the implementation of even simple coherence protocols introduces multiple intermediate states, making the cache lines' state machines exponentially more complex. Memory consistency, on the other hand, imposes the allowed order in which accesses to distinct addresses can be interleaved during execution. Several consistency models ranging in complexity and flexibility have been proposed [6], yet validation of even the simpler ones remains a challenging task.

As multi-processor systems move away from buses toward distributed interconnects with non-deterministic delays, the aforementioned properties become harder to verify, so it can be expected that, in the future, bugs of this type will slip into designs more frequently. To cope with this challenge and prevent costly re-spins, researchers have proposed several runtime validation systems, which can be divided into two groups: checker-based [3, 9, 10] and patching [12, 15] solutions. In the former, the detection is done by a special checker block, which dynamically analyzes the execution trace and detects anomalies or violations of invariants at runtime. Patching-based techniques, on the other hand, use hardware that is programmed at startup with patterns describing the known errors and can detect when the system state matches one of these patterns. The drawback of patching approaches is the need for off-line debugging to develop the proper patterns; their main advantage is that they incur significantly smaller area penalty than checkers.

1.1 Contributions

In this work we present Caspar (CAche Subsystem PAtching and Repair) – a novel patching-based runtime validation solution designed specifically for multi-core processors. Caspar relies on a checkpointing system to record the state of processor's cores and caches. In addition, the technique incorporates on-die cache-event detectors to identify erroneous situations. Each detector is programmed at system startup with *error-condition patterns* that are compared to the local cache line state each time a transition occurs in a local or shared cache. The patterns themselves are created by the manufacturer's support team and are distributed to end-customers when new errors are discovered and debugged. When a memory-related error is detected by Caspar, a recovery and bypass sequence is triggered. The recovery mechanism itself relies on a reduced-complexity operation mode of the system, where only one memory access is performed at a time. This style of execution ensures that there are no interactions between processor cores, therefore, memory coherence and consistency can be guaranteed. Finally, after running the recovery and bypass for a pre-defined period of time, the system resumes regular operation.

One of the key advantages of Caspar is its small performance impact in the absence of bugs. On the other hand, when the processor contains errors, the performance of the system depends on the frequency of error occurrence. Moreover, the event detectors in Caspar occupy less than 0.01% of

978-3-9810801-5-5/DATE09 © 2009 EDAA

the die area (for the OpenSPARC T1 machine), and the majority of the area overhead is due to the checkpoint storage space. Note that, checkpointing can be leveraged for purposes beyond error recovery (*e.g.* post-silicon debugging), potentially amortizing the cost of implementing Caspar.

The rest of the paper is organized as follows: In Section 2 we survey prior work in runtime verification and compare it to Caspar. We then overview our technique in detail and describe its additional potential applications. Then, Section 4 discusses the experimental framework that we used to evaluate Caspar. Finally, Section 5 concludes the paper.

2. PRIOR WORK

Patching and checker-based runtime (or dynamic) verification of microprocessors have been a focus of research investigations in both industry and academia for almost two decades. DIVA [3] is one of the first checker-based solutions for ensuring correctness of execution in a single core processor. A more recent work by Meixner, *et al.* [9] describes another checker-based runtime verification solution for simple processor cores, which uses distributed checkers and leverages compile time information. When an error is detected, the system invokes a backward error-recovery scheme known as SafetyNet [13]. SafetyNet can also be used as the underlying recovery scheme in checker-based solutions for runtime verification of multi-core chips, such as the one in [10]. Caspar differs from these techniques because it relies upon programmable detectors to identify errors, in contrast with non-programmable checkers based on system invariants. Thus, although Caspar requires off-line debugging to produce the error patterns, it incurs significantly smaller area footprint than checker-based solutions, and requires fewer alterations to the chip architecture. In minimize the area impact further, in this work we implemented a checkpointing scheme that is more light-weight than SafetyNet, and does not require augmentation of the chip with additional clock networks for synchronization. This allows our checkpointing scheme to occupy less die area than SafetyNet (6% vs. estimated 8.3% for the target system), while incurring slightly higher performance penalty. It is, however, important to note that the error detection component of Caspar is oblivious of the particular checkpointing scheme, as long as it allows for timely roll-back and recovery after an error is found. Therefore, Caspar can be used with SafetyNet or even more sophisticated checkpointing schemes that incorporate processor I/O [11].

Patching-based approaches for detection and recovery from silicon errors have been introduced by the industry in the late 1990's, when Intel augmented its Pentium Pro CPUs with microcode patching capabilities. A more recent solution for silicon patching was introduced by Wagner, *et al.* in [15], where a small matcher was added to observe the control state of a processor's pipeline and to force a recovery when an error was matched. Note that this solution could only detect an erroneous state of the design, while Caspar's detectors match both state and transition trigger. Another related work [12] presented a more involved solution with multiple recovery schemes; however, both of these techniques concentrate on single core chips. Caspar is different from all of the approaches described above because it is designed specifically for patching the shared-memory hardware in multi-cores. Programmable coherence controllers, such as the one in [4], can also be loosely classified under patching-based solutions, since they allow for the protocol firmware to be updated after system's release. However, this flexibility comes at the cost of slower operation, compared to Caspar, which is specifically designed for error patching.

3. CASPAR OVERVIEW

In this section we overview the structure and the operation of Caspar. We assume a generic multi-core architecture, similar to the one in Figure 1. As shown in the figure, the CPU usually consists of multiple cores, each with a local L1 cache. Controllers manage the caches, satisfying local requests, as well as requests from other cores received through on-chip interconnect. In addition to the local data storage, the cores have access to a larger shared second-level cache. To deploy Caspar in such a system we require the addition of a few hardware modules to the baseline system (hashed blocks in the figure). Note that our solutions is not limited to this architecture and can be used in other multi-core designs with multiple levels of caches, interconnect, *etc.* The majority of the area overhead is due to system-wide recovery, which requires storage space as it is implemented through checkpointing. In Caspar we create a checkpoint of each core's state by logging the values of its architectural registers. For caches we implement a copy-on-write policy and record state and value of a line upon change. Note that checkpoints in Caspar use local storage, eliminating the need to transfer logged snapshots. In addition to checkpoint storage, Caspar augments each cache with a programmable *cache-event detector* – a specialized circuit that compares transitions experienced by individual lines with a pre-defined pattern, thereby identifying memory subsystem errors. If a cache transition matches a detector pattern, Caspar alerts the cache controller, which, in turn, signals the second-level cache to initiate recovery and bypass.

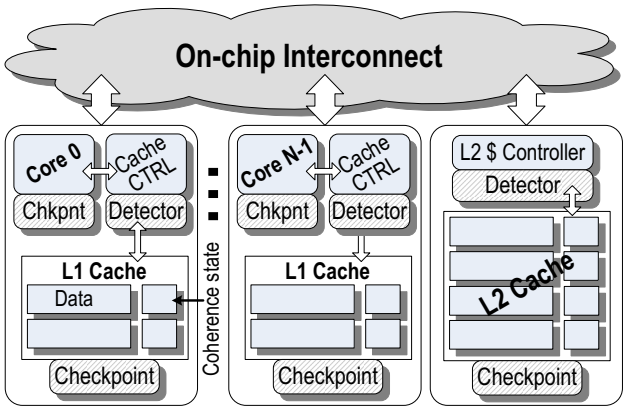

Figure 1: Architecture of a multi-core system augmented with Caspar hardware. Multiple processing elements (cores), each having a private L1 cache, are connected through on-chip interconnect to each other and to an L2 cache. The additional hardware required to implement Caspar is shown in hashed blocks and includes checkpoint storage space and cache-event detectors.

Caspar's operation is organized in epochs, as shown in the schematic of Figure 2. We assume that Caspar event detectors are loaded with appropriate patterns describing erroneous events at startup. At the end of each *checkpointing epochs* the system is synchronized and its state is recorded. *Synchronization* in our framework is initiated by one of the cache controllers (L1 or L2) and forces completion

of all cache transactions in flight. During the synchronization phase no new memory accesses are allowed to enter the system, thus all cache lines reach stable protocol states, after which, at the end of the epoch, a distributed checkpoint is taken. In the second epoch shown in the figure, one of the detector circuits identifies a cache transition corresponding to a known coherence issue and triggers the recovery protocol, which stops program execution and restores the last checkpoint. The L2 cache controller then starts polling individual cores, allowing only one memory access at a time over the entire system, thus ensuring absence of coherence race conditions and enforcing strict ordering of memory accesses. When a pre-determined number of accesses completes, Caspar stores a new checkpoint, and the cores resume operation at full performance. Note that after completion of the last memory operation in bypass mode, the system is again in a stable state, since during recovery only one memory operation was executed at a time.

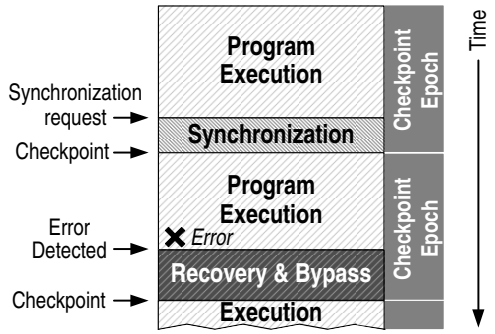

Figure 2: Timeline of Caspar operation. Cache-event detectors are initially loaded with patterns of cache transitions corresponding to errors. In the example no errors occur during the first epoch, which completes with system synchronization and checkpointing. However, during the second epoch an error occurs and is detected, triggering the recovery mechanism. The previous checkpoint is restored, then the system attempts to bypass the error by forcing strict ordering of accesses. After forward progress is made, a new checkpoint is taken and the next epoch commences.

As discussed above, Caspar can be decomposed into three major components: the *checkpointing system* that periodically records the system's state, the *event detectors*, which identify bugs, and the *recovery and bypass mechanism* that allows Caspar to restore valid state and circumvent the error. In the rest of the section we will discuss each of these components and detail their implementation in Caspar.

3.1 Checkpointing

Checkpointing in Caspar is designed to allow fast and efficient system recovery if an erroneous event is detected. As mentioned above, the storage for checkpoints is distributed throughout the system to reside locally with the module whose state it records. The implementation of the core checkpoints is straightforward, since in microprocessor pipelines only the architectural state must be recorded. In Caspar we implement the checkpoint storage for the cores with shadow registers that store the architectural state at each snapshot. Caches and memories are too large to be checkpointed in such a way, so for them Caspar adopts a copy-on-write policy, recording the previous value or the coherence state of a cache line only upon its first change event after the beginning of a new epoch. To this end we augment each cache line with two *modification bits* indicating the modification of a line's data and state during the current epoch. We also create a local log organized as a hardware queue that records address, original state and data of altered lines. The queue and the modification bits are accessed in parallel with the main cache operation and are cleared when a new checkpoint is taken. Then, if a line's state is altered during an epoch, we set the state modification bit in the cache and append the line's address and previous state to the log queue. It is important to note that if only the state is modified the line's data is not copied to the log, allowing for better utilization of the queue storage space. If the block is subsequently altered again, the new change will not be logged, since we are only recording the information necessary to recreate the system's state at the time of the last checkpoint. To restore the last checkpoint state, Caspar suspends the core's operation and performs the following two tasks in parallel: the core's architectural state is restored from the shadow registers and, simultaneously, Caspar traverses the log queue in reverse order, restoring the state and/or data of the modified cache lines from the log while clearing the modification bits. When the traversal is completed, the log queue is cleared.

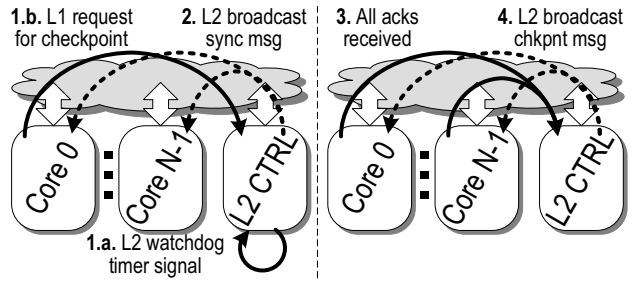

Figure 3: Synchronization for checkpointing in Caspar. Synchronization requests are initiated either by the watchdog timer in the L2 controller (1.a) or by a local cache controller that exhausted its log space (1.b). Following the request, the L2 controller broadcasts a synchronization message to all cores (2), who stop issuing new memory accesses, wait for completion of all pending operations and, finally, acknowledge synchronization (3). As final step, L2 sends a checkpoint message causing architectural state of the cores and local caches to be saved.

Although cache checkpoints in the system are distributed, their recording must be synchronized, ensuring a consistent system state after recovery. In designing our solution we decided to avoid special physical clocks, as required by solutions such as SafetyNet, to synchronize the cores for checkpointing. Such clock networks would require significant routing effort to be implemented in a large multi-core CPU, and incur additional power and area overhead. Instead, Caspar relies on a four-step procedure illustrated in Figure 3. The synchronization is invoked by either a signal from the watchdog timer in the second-level cache (1.a in the figure) or a message from a cache controller that has exhausted its logging resources (1.b). Subsequently, the second-level cache controller broadcasts an explicit synchronization message (2) stopping program execution in all cores. Once all cores have acknowledged completion of all pending operations, the L2 broadcasts a checkpoint message and the state of the chip is recorded as the new checkpoint.

3.2 Detection and Coverage

Caspar is designed specifically for patching errors in the memory subsystem by comparing events (cache state ma-

chine transitions) to pre-loaded patterns describing known bugs. Thus, Caspar can cover a wide class of functional errors associated with unexpected events that are either handled improperly or not handled by the coherence controller. Moreover, as we demonstrate later, Caspar can be extended to detect and recover from memory consistency errors. However, in its current implementation, Caspar is ineffective against bugs that cannot be represented through local state-machine transitions and are caused by invalid request timing, *etc.* The detection feature is implemented with event detectors – small programmable blocks residing at each of the system's caches. Since each coherence transition of a cache line is uniquely defined by the line's state and the input trigger, the error patterns in the detectors consist of two fields, matching the state and the trigger, respectively. Each field is stored as a bit vector, where a 1 in a particular position matches a certain trigger or a state. Note that multiple bits in a pattern can be set to encode combinations of erroneous events.

The diagram of the hardware event detector is shown in Figure 4. During every coherence transition the line's local state and trigger are passed to one-hot encoders, which convert them to bit-vectors describing the event. The vectors are then separately compared with stored patterns on a bit-by-bit basis. The error detection signal, however, is asserted only if the match occurs in both fields. Caspar detectors in general would contain several entries, each identical to the one in the figure, to perform multiple matchings in parallel.

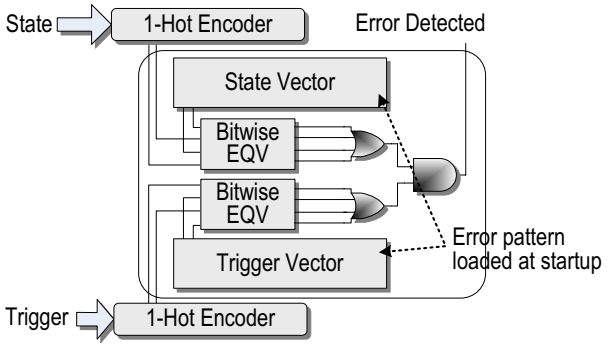

Figure 4: Structure of Caspar hardware event detector. Bug patterns are divided into a state and trigger fields and stored as bit-vectors, where a set bit indicates an erroneous state or transition trigger. During each cache access, the detector encodes the current event as two one-hot vectors and compares them with the stored pattern, asserting an error detection signal only if both match.

We illustrate Caspar's detection in the following example. Assume that a core is performing a store and the line is in the intermediate state "SM" (half-way between "Shared" and "Modified"), when an invalidation message arrives from another core. If the cache's finite state machine was incorrectly implemented, this race could result in a deadlock or coherence violation in a non-Caspar system. However, by programming Caspar with a pattern describing this state ("SM") and trigger (invalidation), we can detect and avoid this error at runtime.

In addition to coherence errors, Caspar framework enables coverage of more complex and subtle issues related to memory consistency. As it was demonstrated in [10], consistency can be ensured if three system-wide invariants are guaranteed: i) uniprocessor ordering, ii) absence of illegal re-orderings and iii) cache coherence. As discussed above, event detectors in Caspar recognize dangerous transitions in the L1 and L2 caches, thus ensuring that the third invariant is upheld. Moreover, as was demonstrated in [10], the checkers for the former two properties can be implemented in efficient small logic blocks (non-programmable) that can signal errors similarly to event detectors. With the addition of such checkers the coverage of our approach can be increased to include bugs which manifest themselves as memory consistency violations.

3.3 Recovery and Bypass

Once an error in the program execution is detected, Caspar initiates a system-wide mechanism to recover from the bug and circumvent it. To return the system to a know-correct state Caspar forces all cores, as well as the local and shared caches, to restore the last checkpoint. Then the system attempts to bypass the error to ensure forward progress. Since errors in the memory subsystem are triggered by interactions of simultaneous accesses, we designed a bypass system to eliminate such race conditions, at a cost of reduced performance. During this phase of operation, we only allow one outstanding memory access in the system at a time, reconfiguring the L2 cache controller to poll cores directly for requests. Note that, in this case, coherence transitions in the system are never interrupted in intermediate states, and cache operations simplify to the level of the underlying coherence protocol. Thus correctness of the bypass operation can be validated using powerful formal techniques. Moreover, since the L2 explicitly orders each core's memory accesses, consistency in bypass mode can also be ensured.

After a pre-programmed number of accesses complete in bypass mode, the L2 controller broadcasts a new checkpointing request, so that the stable state reached in bypass is saved, and the normal execution resumes. Note that many more accesses are usually performed during an epoch's normal execution than during bypass, therefore, it is possible that the offending sequence of instructions is re-executed and triggers an error again. However, in this case the system rolls back to the new checkpoint reached after the last recovery, thus guaranteeing forward progress, so the error is eventually circumvented. Moreover, since the network and message buffers are drained during recovery, the interaction of accesses after bypass is often different than the one that triggered the recovery originally. Thus, it is possible that when the normal operation resumes, the execution of the same sequence does not produce again the erroneous event.

4. EXPERIMENTAL EVALUATION

In this section we introduce the experimental platform used for Caspar's evaluation, and present detailed analyses of bug-finding ability and overheads of our solution. We investigate both the area and the performance overheads due to checkpointing, as well as the recovery penalty for thirteen complex bugs inserted into our baseline design. Finally, we analyze the area impact of Caspar's event detectors.

4.1 Experimental Platform

We simulated the operation of Caspar in a 16-node multicore processor using the Wisconsin Multifacet GEMS architectural simulator [8]. The local L1 caches were 64KB each, while the shared L2 cache had a size of 4MB. The cores were interconnected with a mesh network and used MOESI directory protocol for coherence. Caspar was implemented

inside the GEMS memory tester module (Ruby) as a collection of functions to initiate checkpointing, recovery and bypass procedures, and manage the event detectors, which were inserted into system's caches and directory controller.

Caspar was tested using a set of twenty benchmarks that included ten traces of SPLASH2 benchmarks [16] and additional randomly generated stimuli. The real-world benchmark traces were each 10M instructions long, while the random benchmarks contained 1M memory accesses and varied in degree of data sharing and total address space used.

4.2 Checkpointing Overheads

In our first study we analyzed the overhead of Caspar's checkpointing process. Recall that, to take a checkpoint, the system must synchronize, and thus the execution of some memory accesses must be delayed. Therefore, longer epochs result in a less frequent synchronization. On the other hand, due to the copy-on-write policy for cache checkpointing, longer epochs can only be achieved with larger log sizes, which imposes a higher area overhead. We investigate this tradeoff in Figure 5, where we plot slowdown and area penalty for a range of epoch lengths. The area overhead is calculated as a percentage of cache lines that were modified during the simulation and thus had to be logged. Each data point is computed as an average over the twenty benchmarks. Individual benchmarks deviated slightly from this average: less than 4% for synthetic random programs and less than 1% for real-world applications. As the figure indicates, longer epochs lead to better performance, however, in presence of errors, this will cause more time to be spent in bypass and thus it will incur greater slowdowns. For the subsequent experiments we set the checkpointing epoch length to 8,000 cycles, which translates to roughly 2.4% performance impact and 5.8% area overhead.

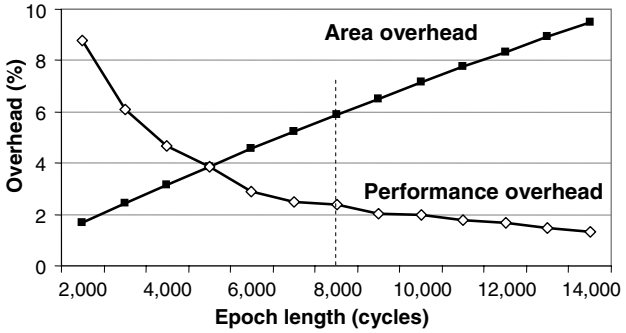

Figure 5: Overhead of Caspar checkpointing. Percent slowdown and area overhead required to provide checkpointing infrastructure over varying epoch lengths.

4.3 Error Resiliency Analysis

In our second study, we inserted thirteen bugs, listed in Table 1, into various modules of the multi-core processor under test. The errors were derived from publicly available errata of commercial multi-core processors and adapted to our evaluation framework. Each bug was associated with a particular state machine transition and described uniquely by the error patterns loaded in Caspar. For each error, we calculated the frequency of occurrence by computing the number of times the state and trigger occurred during normal operation (*i.e.*, with checkpointing and detection disabled). These errors represent complex corner cases of operation of the L1, L2 and directory, when multiple accesses race for resources or an unexpected request arrives when the system is in an intermediate state. Without Caspar enabled, these errors could lead to cache corruption or deadlock, however, when patterns identifying the bugs were loaded into the detectors, all simulations completed correctly.

Table 1: Names, descriptions and frequency of occurrence (occurrences/1M cycles) of errors injected in the experimental system.

Bug name	Description of the error	Freq.
L1 Store race	L1 transitions from S to M while another cache issues a store	0.32
L1 WBack +Load	L1 evicting dirty data while another cache issues a load	0.72
L1 Owner Evict+Load	L1 owner evicting dirty data while another cache issues a load	0.04
L2 Block +Unblock	L2 blocked, waiting for ack while requester issues unblock	34.3
L2 Block +Load	L2 blocked, waiting on directory while L1 issues a load to the line	0.03
L2 Load race	L2 load while another load arrives	0.12
L2 WB	L2 WB while dirty data arrives	2.72
Dir Block +Unblock	Directory blocked on a load while L2 issues unblock	35.1
Dir Clean WB mem	Directory blocked on WB while clean WB forwarded to mem	12.3
Dir Dirty WB mem	Directory blocked on WB while dirty WB forwarded to mem	21.7
Dir Clean WB L2	Directory blocked on WB while clean WB forwarded to L2	0.04
Dir Clean WB L2	Directory blocked on WB while dirty WB forwarded to L2	0.09
Dir Store +Unblock	Directory blocked on a store while L2 issues unblock	34.2

4.4 Caspar Recovery Performance

In addition to the resiliency analysis, we investigated the overhead associated with execution of the 20 benchmarks in presence of different bugs. In this study, we fixed the bypass length to 2,000 instructions, and recorded the fraction of the execution time that the system spent in each of the four phases: normal operation, synchronization for checkpoint, recovery and bypass. The results of the study are shown in Figure 6, where we also plot the average of the thirteen bugs and provide the bug occurrence frequencies for reference. As the experiment demonstrates, for more frequent bugs, a significant portion of execution is spent in recovery and bypass, while, for rare errors, this fraction is negligible. Note that all errors were detected by Caspar precisely, since they were all associated with unique transitions of the cache finite state machines. In this study we observed that sometimes several detections were required to bypass a single bug instance. This was the case when the error occurred late in an epoch and a 2,000-instruction bypass was insufficient to circumvent the problem after the first recovery. In addition, we observed cases where several potential errors, clustered together in program execution, were circumvented with just one bypass. Therefore, longer bypass sequences may be beneficial for bugs that are frequent or tend to occur in groups.

To analyze the impact of the bypass sequence length on the system's performance we conducted an additional study on five errors that varied widely in frequency of occurrence. In this experiment the bypass length varied from 500 to 5,000 instructions and the overall performance slowdown of the system was recorded (see Figure 7). As you can see, a longer bypass was beneficial for frequent errors, primarily because it guaranteed that the error would be circum-

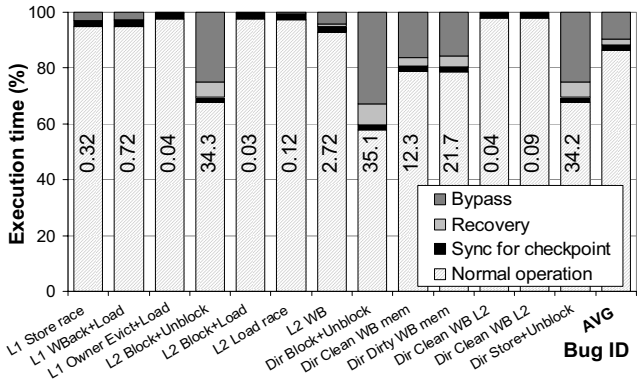

Figure 6: Execution time breakdown. Percentage of time the system spends in the four execution phases (normal operation, checkpointing, recovery and bypass) for each of the thirteen errors.

vented on the first attempt and there was a higher chance that multiple errors would be covered by a single bypass. On the other hand, we noticed the opposite trend for rare bugs (*L1 WBack+Load* and *L2 Block+Load*). In these cases, errors instances were significantly farther from each other, thus a longer bypass did not reduce the number of recoveries and only incurred higher performance overhead. From this study we conclude that Caspar's fixed-length bypass strategy is not optimal for all bugs. Indeed, the system could be improved to load the number of instructions to run in bypass together with the bug pattern, or calculate the bypass length dynamically based on where the last error was detected from the beginning of the epoch.

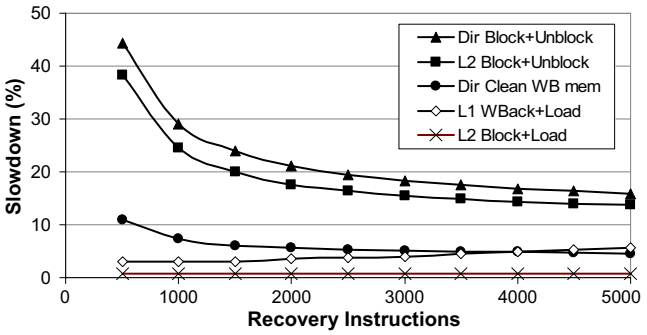

Figure 7: Performance vs. bypass length. For frequently occurring errors, a longer bypass is more beneficial, for it allows multiple bug instances to be covered together. For rare bugs, on the other hand, a longer bypass results in larger penalties due to longer execution at lower performance.

4.5 Event Detector Area Overhead

Finally, we investigated the area overhead of Caspar's event detectors. To this end we implemented the detectors for the L1 and L2 caches in Verilog HDL and synthesized them using the TSMC 90nm technology. The area of the L1 and L2 cache detectors was found to be 984.2μm^2 and 2,422μm^2, respectively. In comparison, an OpenSPARC T1 chip, which has eight cores with private caches and a shared 4-bank L2 cache, has an approximate area of 378mm^2 [7]. Thus, placing event detectors in each private cache and in each L2 bank incurs an area overhead of 0.005%, a negligible penalty compared to the area of the checkpointing log storage analyzed in Section 4.2. Considering that the area overhead of the checkpointing is calculated as a percentage of the area of the caches, we find that in the OpenSPARC T1 design the checkpointing logs and detectors of Caspar will occupy less than 3% of the total die area.

5. CONCLUSION

In this work we presented Caspar – an effective solution for in-the-field patching and repair of the memory subsystem in modern multi-core processors. Caspar relies on periodic system checkpointing and hardware event detectors programmed with descriptions of known erroneous transitions of the system's state machines. When an error is detected during operation, Caspar stops the execution, recovers the correct state, and attempts to bypass the bug by enforcing race-free operation. Our experimental results demonstrate that Caspar's checkpointing scheme incurs little performance and area penalty (2.4% and 5.8%, respectively, for the target system in our experiments). Thus, with no errors present in the design, the impact of our solution is kept quite small. When errors do occur, recovery can potentially become expensive; however, we found experimentally that with low error frequencies (which is expected given that the system has undergone heavy design-time verification) the overall performance penalty is also small. These features allow Caspar to work as an effective patching solution, protecting even the most complex of today's designs.

6. REFERENCES

[1] Intel(R) Pentium(R) Processor Invalid Instruction Erratum Overview, July 2004. www.intel.com/support/processors/pentium/sb/cs-013151.htm.

[2] *Revision Guide for AMD Family 10h Processors*, Apr. 2008. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/41322.PDF.

[3] T. M. Austin. DIVA: A dynamic approach to microprocessor verification. *Journal of Instruction-Level Parallelism*, 2000.

[4] L. A. Barroso, K. Gharachorloo, M. Ravishankar, and R. Stets. Managing complexity in the piranha server-class processor design. In *Workshop on Complexity-Effective Design*, 2001.

[5] S. Bell et al. TILE64 processor: A 64-core SoC with mesh interconnect. In *Proc. ISSCC*, pages 88–598, Feb 2008.

[6] D. Culler, A. Gupta, and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[7] A. Leon, K. Tam, J. Shin, D. Weisner, and F. Schumacher. A power-efficient high-throughput 32-thread SPARC processor. *IEEE Journal of Solid-State Circuits*, 42(1), Jan. 2007.

[8] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *ACM Computer Architecture News*, 33(4), 2005.

[9] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. *IEEE Micro*, 28(1):52–59, 2008.

[10] A. Meixner and D. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proc. DSN*, pages 73–82, 2006.

[11] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: efficient handling of I/O in highly-available rollback-recovery servers. In *International Symposium on High-Performance Computer Architecture*, Feb. 2006.

[12] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro Special Issue*, 27, Jan. 2007.

[13] D. Sorin, M. Martin, M. Hill, and D. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. ISCA*, 2002.

[14] S. Vangal et al. An 80-Tile sub-100-W TeraFLOPS processor in 65-nm CMOS. *Journal of Solid-State Circuits*, 43(1), 2008.

[15] I. Wagner, V. Bertacco, and T. Austin. Using field-repairable control logic to correct design errors in microprocessors. *IEEE Transactions on Computer-Aided Design*, 27(2), Feb. 2008.

[16] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. ISCA*, 1995.