2011 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems

An Architecture to enable Life Cycle Testing in CMPs

Rance Rodrigues, Israel Koren and Sandip Kundu
Department of Electrical and Computer Engineering
University of Massachusetts at Amherst
e-mail: {*rodrigues, koren, kundu*}@ecs.umass.com

***Abstract*— CMOS wear-out mechanisms such as time dependent breakdown of gate dielectrics (TDDB), hot carrier injection (HCI), negative bias temperature instability (NBTI), electromigration (EM), and stress induced voiding (SIV) are well documented in the literature. Often the onset of wear-out is gradual, with initial manifestation as delay defects that result in timing errors. This motivates the need for online testing. The combined effect of dynamic reconfiguration such as voltage and frequency scaling (DVFS) and signal integrity issues coupled with aging related wear-outs complicate *a priori* selection of test vectors, further favoring online testing. Traditional online test techniques such as Double and Triple Modular Redundancy (DMR and TMR) pose severe area and power overheads. In this paper we propose an architecture to assist online testing in a Chip Multiprocessor (CMP) based on execution path recording. Since in practice, core utilization in CMPs is low, we can use the idle time of cores opportunistically to run test threads that mimic functional threads. The initiation, termination and comparison of test results is performed by a dedicated, simple and functionally limited small core that we call the Sentry Core (SC). The sentry core is hidden from the OS and has the ability to monitor and interrupt the general-purpose cores. Upon interrupt, the general-purpose core can send data to the sentry core. To detect errors, the SC initializes the general-purpose cores and collects signatures from hardware monitors (that compact execution traces) and compares them against duplicate test threads, obviating any need for cycle by cycle comparison. Major benefits of the proposed solution include: (1) online testing with minimal area overhead, (2) scalability, and (3) testability throughout the life cycle of a CMP. Experimental results show that the proposed scheme is capable of detecting 87% of the faults injected into the processor at an area overhead of less than 3% of the target CMP.**

Keywords: online fault detection; microprocessor test; opportunistic test; low-cost test

I. INTRODUCTION

Aggressive scaling of technology has led to ever smaller transistors [4]. As transistors become smaller and interconnects become thinner, current and power densities increase, which in turn increase the operating temperature of the chip. Increased current density accelerates CMOS aging mechanisms such as electromigration [27], ultimately leading to defects [4],[6]. Similarly, increased temperature triggers device degradation through effects such as NBTI [29]. Increased electric field causes device degradation and ultimately device failure due to TDDB and HCI [29]. Together, CMOS aging mechanisms pose significant challenges with scaling.

The International Technology Roadmap for Semiconductors (ITRS) predicts that by 2020, a monolithic IC will contain 1000 billion transistors [32], potentially integrating 20 to 64 processor cores in a chip multiprocessor. This poses both challenges and opportunities. The challenges stem from device reliability trends that project a dismal picture for yield and performance with scaling, while the opportunities stem from inherent resource redundancies in such chip multiprocessors.

The device aging effects unfold over time. Initially, aging defects manifest themselves as delay faults [26],[30]. Later, they become gross defects resulting in catastrophic failures. Historically, chip manufacturers have countered the performance degradation that occurs over the life of a product via performance guard-banding. For example, several researchers have shown that with NBTI, device performance becomes slower by 3-5% [33]. Thus, a conservative 5% frequency margin at the time of performance sort testing after manufacturing could account for such degradation that will occur over the life of the product. This solution however is coming under pressure for multiple reasons. First, the magnitude of the worst case degradation is increasing, making the ultra-conservative guard-band negate the benefit of newer design and scaling. Second, actual degradation itself is non-uniform. For example, in a many core chip, the average core utilization may be low. If some cores remain constantly active while other cores are idling, the active cores may age faster. Similarly, within an active core, depending on the actual workload, some units may age faster than others. These factors argue against a static performance guard-banding.

The alternative to static performance guard-banding is dynamic performance tuning. Dynamic performance tuning requires the determination of the maximum allowed frequency (also known as F_{MAX} testing) in the field. Since traditionally F_{MAX} testing is done with a tester, this poses a challenge. In this paper, we overcome this challenge by online testing. A CMP has many cores and therefore, two cores can execute the same program and continuously check their results for online error detection. However, such a simple cycle-by-cycle checking is impossible in practice. First, even identical programs executing on identical cores (and starting at the same cycle) will begin to diverge in their operation as memory controllers must serialize the memory requests and processors may idle differently. Second, when interrupts, traps, memory page allocations via OS interactions are taken into considerations, the programs tend to diverge further in time. Third, if the two cores are not identical, such divergence becomes acute. Non identical cores may be an artifact of design such as asymmetric chip multiprocessors (AMP) or may be the result of a core being assigned a different frequency than the other core. In this paper, we present a solution to this problem of online checking where the programs are not

1550-5774/11 $26.00 © 2011 IEEE
DOI 10.1109/DFT.2011.26

checked on a cycle by cycle basis. Some researchers have investigated these problems previously. Their proposed solutions are reviewed later in this paper, in Section V. As will be explained, none of the previous approaches provide a complete solution to the above problems.

Our solution is based on incorporation of an additional core called Sentry Core (SC), a unit whose goal is to assist fault detection in a CMP (Figure 1). The SC is akin to service processors that have been used in the past in main-frame computers [19],[20] or watchdog monitors [34], but also differs in significant ways. The most important differences are: (i) embedding the SC in the CMP allows the internal states of the CMP cores to be accessed, and (ii) the SC is an active participant in dealing with traps and interrupts. The SC is a small and simple core, so simple that it can be assumed to remain functionally correct throughout the life of the CMP. Since the SC does not execute functional code and its tasks do not require high performance, large performance guard-band for SC is not a concern. Similar assumptions have been used for IBM service processors and the DIVA checker [1].

Whenever cores are found to be idle, the SC initiates test routines on the cores, and then captures and compares the resulting responses to detect faulty behavior. The central idea in this paper is to have the SC capture signatures of the program execution that reflect both the control flow and the data execution, which are then compared against responses obtained on-line from another core in the CMP. More specifically, we collect two signatures for the executing program. The first one compresses the program counter values associated with each committed branch instruction into a multiple-input signature-register (MISR). The SC performs the tasks of initializing the MISR, collecting the signature of a fixed number of executed branches and comparing against the reference obtained dynamically from another core running the same thread. Such a comparison reveals errors in the control flow of the program. The second signature is associated with store instructions. Whenever a store instruction is executed, a write occurs into a memory address. In practice, such writes will modify the cache but may never get written back into the main memory before the program crashes. To avoid the loss of such information, we collect a signature of the data value and the data address in a MISR on every committed write instruction. Since the signatures are collected on committed instructions, speculative execution has no effect on these signatures. The virtual address of the branch/store instructions is used to compute the signature. Hence, the same signatures will be generated for two fault-free cores running the same code segment. Lastly, while such monitoring is in progress, traps and interrupts will be routed via the SC and any exception will halt the program execution allowing the SC to access the appropriate signatures. When a fault occurs, the signatures obtained from the two cores will differ and the fault will be detected.

The benefits of the proposed solution are: (i) online testing with minimal overhead, (ii) scalability and (iii) lifecycle testability.

To validate the proposed approach we conducted experiments using the SESC simulator [7] and used eight benchmarks from the SPEC 2000 suite [8]. We chose these benchmarks and not specifically engineered test routines to show the potential benefits of the SC regardless of the executed software. The development of test code to accelerate fault detection is not the focus of this paper and is a part of future research. In the experiments, faults were injected to result in a faulty behavior in a 4-core CMP and the resulting fault detection latency and coverage were measured. The relationship between the fault detection latency and the checking interval (time between signature checks) was also analyzed. Our results indicate that even though the SC may add an area overhead of up to 3% for the target system, the rich testing functionality it provides makes it an attractive approach.

The rest of the paper is organized as follows. In Section II, the sentry core architecture, the resulting overhead and the fault detection strategy are discussed. The evaluation framework is described in Section III and results are presented in Section IV. Prior research is summarized in section V followed by limitations of our approach and final conclusions in Sections VI and VII, respectively.

II. PROPOSED SOLUTION

Our scheme is built around the SC which verifies the fault-free operation of the general-purpose processor cores. In this section, we describe the functionality and hardware overhead of incorporating an SC in a CMP. The approach followed to detect faults in the CMP is then presented.

A. *Sentry Core*

The SC is a small and simple core with the objective of enabling quick fault detection in a CMP. The SC needs to have the capability to initialize the test as well as collect and compare signatures and detect faults if any. To this end it is augmented with a variety of features described next.

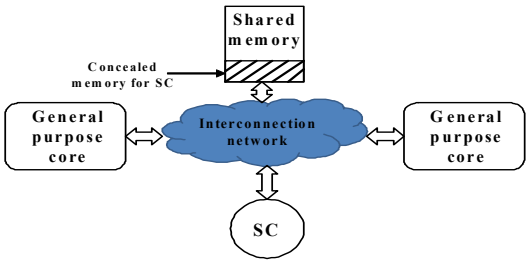

Figure 1: Sentry core (SC) with a dual core processor.

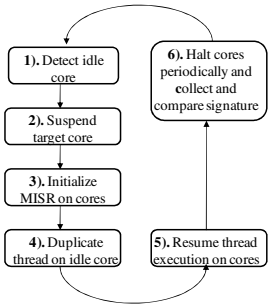

Figure 2: Sequence of steps followed SC for test scheduling.

1) Control functions

To assist in fault detection, the sentry core supports the following operations:

1. **Detect idle cores**: SC has the capability to detect if cores are idling.
2. **Initialize MISR**: initialize the MISR for each core participating in the test.
3. **Duplicate**: trigger a DMR/TMR (Dual/Triple Modular Redundancy) configuration by replicating a process and executing it on two or three cores.
4. **Collect MISR signature**: collect and compare signatures periodically.
5. **Suspend**: halt one or all the processor cores to analyze their state.
6. **Resume**: resume the operation of a halted core(s).
7. **Terminate**: terminate a process on a core.

We have listed only a subset of the many possible functions that a sentry core can provide for fault detection. This functionality can easily be added to any off-the-shelf processor that fits the description of the SC by extending the Instruction Set Architecture (ISA). In most processors, the opcode field size is such that it is extensible. In [23], an example of extended ISAs is described listing multimedia instructions added to the ISA of general-purpose microprocessors. The added instructions will be used by the SC to control the cores in the multicore. The steps followed by the SC to initialize testing are shown in Figure 2.

When the cores are halted for signature comparison, the state of the cores must be saved so that execution may later resume. After saving the state, the cores provide the current control/data signatures to the SC (may be done via a write to a shared memory location) and then resume execution when permitted to do so. The routine to save the MISR state to a shared memory location may be the same as that followed for interrupts. During interrupts, the program state is saved; the interrupt handling routine is executed and the core then resumes operation after state restoration. A similar procedure is followed during context switch while running multithreaded applications. The saving of the MISR state to memory may thus be implemented as an interrupt handling routine or context switch in the CMP. Thus, the overhead incurred by the SC intervention for signature comparison is similar to that of a context switch. In [25], Li et al. quantify the effect of a context switch considering processors with various cache and array sizes. They report that the direct overhead of a context switch can range from a few hundreds to a few thousands cycles depending on cache size and configuration. We assume 1000 cycles as the context switch overhead for the target system configuration. Hence, we conclude that the overhead for a SC operation when using already available mechanisms is about 1000 cycles. In general, a dedicated design for the communication between the SC and the general-purpose cores will likely yield a far lower overhead (10 – 100 cycles) but it may incur a small hardware overhead. In the rest of this paper we assume the presence of the interrupt based scheme due to its simplicity.

2) Hardware considerations for the SC

The SC provides rich fault detection functionality in a CMP with a small hardware overhead. We already saw that for communication between the SC and the other cores there is no area overhead when using interrupts. To estimate the overhead of incorporating the SC itself in a CMP, we consider for example, the EV4 (Alpha 21064), and EV6 (Alpha 21264) cores. The functionality of the EV4 is sufficient to satisfy the requirements of an SC for a CMP comprising of EV6 cores. The EV4 occupies about 11% of the area of an EV6 core [24]. Hence, for a dual/quad/eight core CMP, the area overhead due to the SC reduces to 5.5/2.75/1.375%. This is an acceptable overhead for the added functionality considering that DIVA [1] adds a 6% overhead. While making this comparison in area, it is important to note that the SC scheme is a highly scalable solution unlike DIVA which incurs a fixed overhead for every core in the CMP. A single SC can service a quad, eight or even a sixteen core CMP. However, as the number of cores serviced by the SC increases, the time slice that each core gets for service reduces. For many core systems, additional SCs may be incorporated such that there is one SC for every *m* cores in the CMP. We plan to evaluate this in the future. The above is just an example to illustrate the size of the overhead. In reality, a customized (rather than an off-the-shelf) SC design may have far lower area overhead (< 1%). Incorporation of an SC results in heterogeneity which increases the design time, but heterogeneity is no longer a novel concept [1],[24] and hence may be considered an acceptable practice.

B. Fault detection strategies

For detecting faults a reference is needed for comparison. One way to generate the reference is the same as is followed in DMR. However, as described earlier, the SC will initiate DMR only when two cores are idling thus overcoming the drawbacks of continuous DMR. Whenever idle cores are found, the SC copies the program state and program data of one core to another core. Both cores execute the same program for a fixed number of branch instructions as determined by the SC. When the number of branch instructions executed by the core reaches its pre-specified limit (a programmable parameter), an interrupt is generated and the program is vectored into a wait state as described earlier. The SC can then query the signatures generated by the executed branch instructions as well as the memory write operations. If a mismatch is encountered a fault is detected. This scheme will be effective for heterogeneous multicores as the underlying hardware in the cores is different. The scheme will also work for homogeneous multicores as different cores have different susceptibility as described earlier.

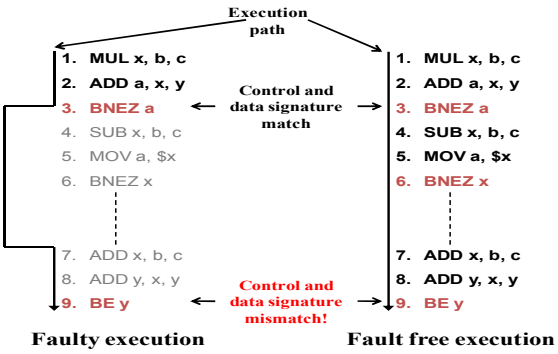

Figure 3: An illustration of the fault detection mechanism.

As an example, consider a heterogeneous multicore system like the one considered by Kumar et al. in [24]. Their proposed multicore consists of Alpha cores of different technologies, but they all have the same ISA. To generate the comparison signatures for a specific test code in this case, the smaller EV4, which is far less complicated than the other EV5/EV6 cores and has been already verified, may be used. The same test code can then be run on the more complex EV5/EV6 and signatures can be compared. A discrepancy will indicate the likely presence of a fault.

C. *Fault detection algorithm*

The SC supports fault detection in CMPs by means of special control signals. It can pause, resume, and suspend the operation of the cores in the CMP. The fault detection procedure uses the control signals mentioned in Figure 2 and the test strategies described in subsection IIB. As described earlier, the SC copies the program state of the core to be tested onto another core (an idle core). The number of branch instructions after which the signatures are to be compared is set (detailed experiments on the choice of this variable to follow) and the cores then begin execution. After committing *n* branch instructions, the branch counter resets to zero and an interrupt is generated. Both cores then execute the interrupt handling routine and store the MISR signature to the shared memory. The SC then collects and compares the signatures. If they match, execution is resumed on each core for other *n* branch instructions. If however, the signatures do not match, a fault is detected.

Figure 5 shows an example of running a test code on two cores, the execution of which is faulty on one core and fault-free on the other. The execution path taken is indicated by the solid lines with arrow heads. It can be seen that due to a fault in the computation of the value stored in register ***a***, the condition checked by the branch instruction evaluates to true for the faulty core, and false for the fault-free core. As a result, the faulty core now executes along a different path. The two cores then encounter different branch instructions and by comparing the signatures of the execution traces on the two cores the SC can detect the fault. Similarly, if there was a fault while storing a value into register ***a***, the store signature generated by the two cores would differ indicating the presence of a fault.

III. EVALUATION FRAMEWORK

In our experiments we used the SESC architectural simulator [7] after modifying the code to allow injection of faults to cause an erroneous behavior in the control and data paths. We used eight SPEC CPU 2000 [8] benchmarks as test codes for the sake of demonstrating the effectiveness of our approach. We assume that the SC is incorporated in a symmetric quad core CMP in which one core is faulty. The system parameters of general-purpose cores in the 4-core CMP are shown in Table 1. The benchmarks used were *equake*, *ammp*, *swim*, *wupwise*, *applu*, *gzip*, *gcc* and *mcf* and were chosen so as to be representative of several classes of benchmarks (FP/INT/load/store/Branch intensive) that would exercise different units within the processor. The instruction distribution for each benchmark (after skipping 5 billion instructions and then executing them for 10 million instructions) is shown in Figure 4.

TABLE I. SYSTEM PARAMETERS

Frequency	2 Ghz
Fetch/Issue/Retire	4/4/4
ROB size	128
ISQ size	80 INT, 40 FP
Branch Prediction	Hybrid: local bits 2, BTB 4096 entries RAS size 64, Replacement policy LRU
Functional units	2 FP and 4 INT ALU 1 each of FP/INT MUL, DIV
Registers	104 INT and 80 FP
L1-D/I cache	64K, 8-way, 1cycle
L2 cache	2M, 16-way, 10 cycle

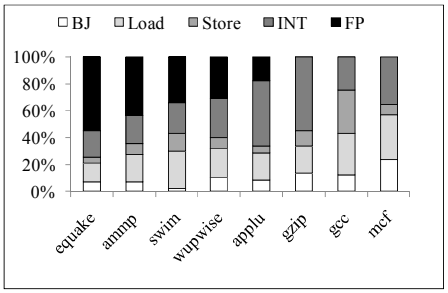

Figure 4: Instruction distribution for the eight benchmarks

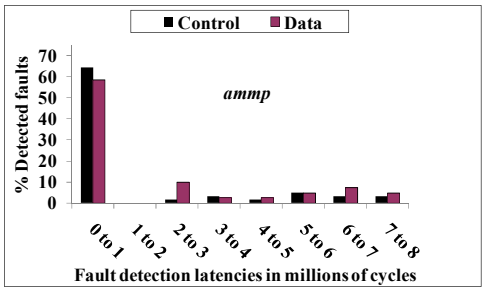

Figure 5: The distribution of the detected faults as a function of the detection latency for *ammp*

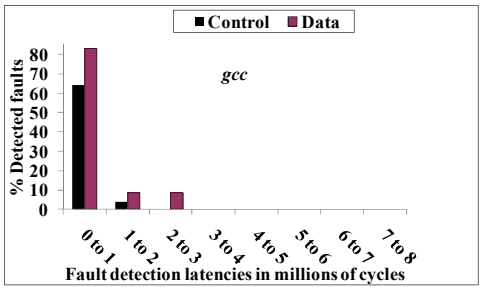

Figure 6: The distribution of the detected faults as a function of the detection latency for *gcc*

Since we use a performance simulator, injecting faults into the system is implemented through bit flips in the data structures used to simulate the architectural components. For example, when an instruction is retired, a bit in the Reorder Buffer (ROB) data structure is set to indicate that. If a fault is injected into that bit of the ROB entry, other instructions waiting for this instruction to complete will never resume execution. There are also cases where the injection of a fault may result in multiple faults due to the way the simulator operates. To cause a faulty behavior during a store operation, we either inject a single bit fault in the data register or inject a fault into the address register, resulting in wrong data value or address. In each benchmark run we have injected 100 data and control faults.

IV. RESULTS

In this section we present the experimental results for fault detection latency, and fault coverage. We first present the range of latencies experienced before detecting the injected control and data faults. The fault detection latency is measured from the time at which the fault is injected until it is detected. The average fault detection latencies for control and data faults are shown in Figures 5-7 to illustrate the relationship between the test routines and the average detection latencies. Continuous cycle by cycle monitoring of the other cores by the SC will result in prohibitive overhead (~1000 cycles per control operation as described earlier). To avoid that, the SC does the checking after a set number (n) of branches are committed. The effect of varying the value of n on the fault detection latency is studied below. Based on these experiments, fault coverage results when using an SC checking interval of 100K branches are presented.

A. Fault detection latencies

The observed fault detection latencies are shown in Figures 5-7. We have only included results for *ammp, gcc* and *mcf* as these were found to be interesting. These results include both data and control faults for a fixed checking interval of length n =100K committed branches. It can be seen that for almost all benchmarks a considerable percentage of the injected faults (about 60% on average for control and 55% for data faults) are detected within 1 million cycles of execution. Control related faults manifest themselves faster than data related faults and hence are caught earlier. There are only a few faults which are detected at later stages and a majority of those are data related faults.

By observing the latencies at which the majority of the faults are detected and the instruction distributions (Figure 4) of the benchmarks, it can be seen that there is a correlation between the two. For an injected control fault to manifest itself during the program execution, the corresponding control must be exercised. For example, a fault in an integer issue queue entry will manifest itself only if that entry is used, i.e., the program should have sufficient integer (INT) instructions. The same holds for floating-point (FP) instructions. Similarly, for faults inserted in the load/store queues to be visible, the load/store queue must be used frequently enough. We found that on average, control related faults are caught earlier for the benchmarks that have a reasonable percentage of FP and INT instructions (e.g., *wupwise*). Having just INT (FP) instructions (e.g., *gzip*, *equake*) may mean that faults injected in the FP (INT) execution path may not be exercised. Also, memory intensive benchmarks spend considerable time waiting for the memory operations to complete and as a result, control faults do not manifest themselves fast or not at all. Still, having a reasonable percentage of INT and FP instructions may not guarantee increased coverage as often programs run loops which execute the same type of instructions repeatedly thus potentially missing the ones that can exercise the fault. In addition, since our scheme collects signatures only after n branches are committed, a higher frequency of branches in the test routine results in early fault detection (*mcf*, *gzip*, *wupwise*). If the branches are sparse, the fault may be detected after a very long latency (*swim*) and the fault may even go undetected. Hence, for control related faults, the frequencies of INT/FP and branch instructions play a major role in determining the fault detection latency. A similar explanation exists for the data related faults. Since these faults will be exercised only when data is being stored (to the caches), the larger the number of store instructions the higher is the chance to exercise these faults (*gcc*). Here too, having frequent branches may reduce the fault detection latency (*gzip*). The above discussions apply to the plots showing detection latencies (Figures 5 - 7) and Figure 8 that shows the average control and data fault detection latencies for all benchmarks.

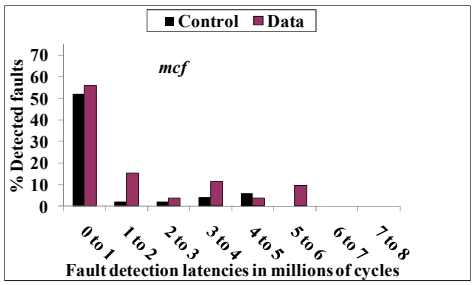

Figure 7: The distribution of the detected faults as a function of the detection latency for *mcf*

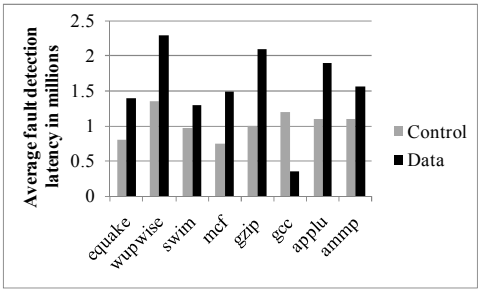

Figure 8: Average detection latencies for data and control faults for all eight benchmarks

B. *Fault coverage*

Just as the instruction distribution plays a role in determining the detection latencies, it also impacts the fault coverage. The fault coverage for various benchmarks is shown in Figure 9. It can be seen that the larger the utilization of the different execution paths (FP/INT), the higher is the control fault coverage (*wupwise*, *ammp*). The same holds for the coverage of data faults (*gcc*). An important observation is that we were able to catch just 82% of the control faults as compared to 92% of the data faults on average. The reason for this difference is that control faults were mostly caught early or not caught at all. This happens as (i) control faults if exercised, manifest themselves sooner, and (ii) some of the benchmarks are either FP (*equake*) or INT (*gzip*) or memory intensive (*gcc*) and some have very few branches (*swim*), and hence were not exercising all the injected faults as explained earlier. Data faults were caught more often but with higher latencies. The most notable result is that for *gcc* where 100% coverage of data faults was achieved due to about 30% store and 10% branch instructions found in the mix. Overall, the scheme was able to detect 87% of the (combined control and data) faults using standard benchmarks rather than specifically engineered test routines, which greatly increases the confidence in the ability of the proposed scheme to detect faults online.

C. *Effect of number of branches committed before signature comparison*

The SC collects and compares signatures of the execution traces only after a set number *n* of branches are committed. It is expected that the smaller the interval, the better it is for our scheme. But if the signatures are compared too often, this results in a higher overhead (~1000 cycles for each comparison) for checking. To this end we investigated the effects of various checking intervals. The average fault detection latency when using checking intervals of 1K, 10K, 100K, and 1000K committed branches is shown in Figure 10. Each bar in the figure shows the average for both control and data faults. The net detection latency is the fault detection latency with zero checking overhead. Hence the final detection latency is the sum of the net detection latency and the overhead (total height of the bars in the figure). It can be seen that for the smaller checking intervals even though the net detection latency with no checking overhead is small, the checking overhead is high due to very frequent checks. For larger checking intervals, even though the net detection latency increases, there are fewer checks. Hence the final fault detection latency with checking overhead is small. However, this trend continues only until a 100K interval length. After that, when using for example, a 1000K interval length, even though the checking overhead is small (since there are very few checks) the fault detection latency is very high. We therefore, used in our experiments 100K as the interval length.

V. PRIOR RESEARCH

With aggressive technology scaling, aging defects afflict processors with progressively worse delay and catastrophic faults. As a result, fault detection and correction schemes have been a topic of considerable interest. Previous approaches may be classified into those that target certain structures in a processor ([5], [10], [13], [14], [15]) and those that target the entire processor ([1], [2], [3] ,[9], [11], [12], [31], [35], [36]). Of these a few of them are directly comparable to our approach.

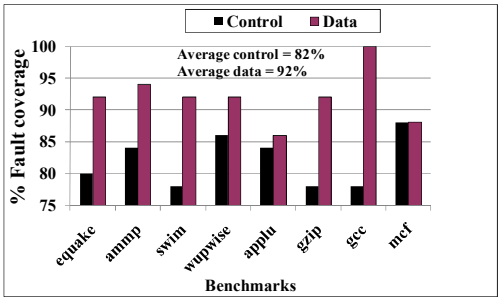

Figure 9: Data and control fault coverage for the benchmarks

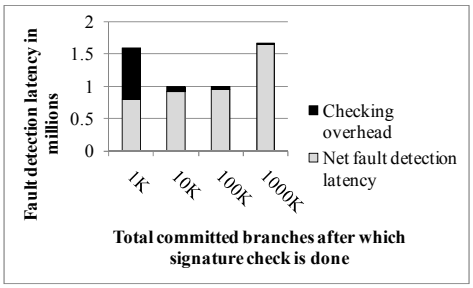

Figure 10: Effect of checking interval on fault detection latency. Data shown is averaged for all considered benchmarks

In [5], Bower et al. presented a scheme to detect and tolerate faults in array structures of microprocessors. A similar scheme was presented by Rodrigues et al. in [15]. Fault detection in integer ALU execution units is proposed by Abella et al. in [10]. Self test for register data flow is proposed by Carretero et al. in [13]. Online periodic testing for the floating-point units of a processor considering various ISAs is explored in [14]. However, these schemes only protect certain structures of the processor and do not provide chip wide coverage.

Chip wide error detection schemes have also been proposed. In [9], Shyam et al. protect stages of the pipeline using BIST techniques. Meixner has proposed Argus, a dynamic verification scheme for fault detection in simple cores [11]. Li et al. [12] use high level symptoms with system restoration and re-execution on another core to detect faults. Use of reconfigurable fabric for testing cores is proposed by Deng et al. in [22].

A few of the chip wide error detection approaches have similarities with ours. Austin has proposed the DIVA checker [1] in which a small core is augmented to check for computation correctness of its companion core. Whenever the results from the two cores differ, the checker core commits its result and the pipeline of the larger core is flushed. However, each core requires a DIVA core for error detection which is not the case with our proposal where multiple cores may share a single SC. Replication of pipeline stages [35] and complete replication of core execution for fault tolerance has been explored via DMR/TMR in [31]. These approaches pose very high area and power overhead (200/300%). Our SC based approach reduces these overheads by initiating DMR only when cores are idle. Redundant Multi Threading (RMT) approaches have also been proposed [2],[3], in which a logical thread is run as two physical threads. One of the threads is leading while the other is trailing and the leading thread provides certain inputs to the trailing thread. A difference in execution indicates the presence of a fault. The states of the two threads are compared while our SC based solution only compares the signatures. The amount of state information compared in RMT is important as it determines the overhead. Smolens et al. [36] proposed a solution in which the fingerprint of instructions between checkpoints is compared for error detection. Here comparison of states is assumed to be done by an error-free core. Our SC based solution collects signatures of committed branch instructions and then compares them at regular intervals to detect faults. Since the SC is responsible for signature comparison, there is considerably lower probability of an error during this comparison.

Thus, even though there are similarities between our approach and the previous work, our scheme overcomes the drawbacks of other approaches.

VI. LIMITATIONS

Although the SC enables flexible online testing, there are limitations to this approach. Adding an SC to the CMP architecture increases the design verification effort. However, as the SC is small and simple, it can be easily verified. Also, since we use signatures, there is always a chance of aliasing. However, since signatures are 32-bit long, the probability of aliasing is small.

VII. CONCLUSIONS

We proposed a new approach to online testing of multicore processors running multithreaded programs. The main problem with existing solutions is that they either incur high overhead (DMR/TMR) or are not comprehensive in testing all parts of the processor. We proposed a diminutive core called sentry core to provide the basic functions of initiating, halting and querying processors. The added sentry core adds less than 3% in area, but enables a rich set of testing features. In theory, the signatures of instruction path and memory writes should provide complete coverage barring masked faults. Simulation results validate that when the faults are triggered by the executing program, they are detected. Further, the latency of detection from trigger to detection is small. This is important because with a small latency a small test code will suffice for fault detection. When using a 100K committed branches as checking interval, our approach was able to detect about 82% of the injected control and 92% of the data faults with an average detection latency of about 1 million cycles for control and 1.5 million cycles for data faults (see Figure 8).

ACKNOWLEDGEMENT

This work has been supported in part by a grant from the GRC and the NSF (Grant No 1985.001).

REFERENCES

[1] Austin, T.M., "DIVA: a reliable substrate for deep submicron microarchitecture design," Proceedings of the 32nd Annual International Symposium on Microarchitecture, MICRO-32 , pp. 196-207, 1999.

[2] Rotenberg, E., “AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors,” Proc. International Conference on Dependable Systems and Networks (DSN) 1999.

[3] Reinhardt, S.. Mukherje, S. S., “Transient Fault Detection via Simultaneous Multithreading,” Proc. IEEE/ACM International Symposium on Computer Architecture (ISCA), 2000.

[4] Srinivasan, J., Adve, S.V., Bose, P., Rivers, J.A., "The impact of technology scaling on lifetime reliability," Proc. International Conference on Dependable Systems and Networks, pp. 177-186, June- July 2004.

[5] Bower, F.A., Shealy, P.G., Ozev, S., Sorin, D.J., "Tolerating hard faults in microprocessor array structures," Proc. International Conference on Dependable Systems and Networks, pp. 51- 60, July 2004. doi: 10.1109/DSN.2004.1311876.

[6] Borkar S. Y., “Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation,” IEEE Micro, Vol. 25, Issue. 6, pp. 10-16, Nov.-Dec. 2005

[7] Renau, J.; et al., SESC Simulator, January 2005. http://sesc.sourceforge.net

[8] The Standard Performance Evaluation Corporation (Spec CPI2000 suite). http://www/specbench.org/osg/cpu2000

[9] Shyam, S., Constantinides, K., Phadke, S., Bertacco, V., and Austin, T., "Ultra low-cost defect protection for microprocessor pipelines," Proc. SIGPLAN No. 41, 11, Nov. 2006, pp. 73-82.

[10] Abella, J., Vera, X., Unsal, O., Ergin, O., Gonzalez, A., "Fuse: A Technique to Anticipate Failures due to Degradation in ALUs," Proc. 13th IEEE. International Symposium on On-Line Testing, IOLTS 07, pp. 15-22, July 2007.

[11] Meixner, A.; Bauer, M.E.; Sorin, D.J.; "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," Proc. 40th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2007.

[12] Li, M-L., Ramachandran, P., Sahoo, S.K., Adve, S.V., Adve, V.S., Zhou, Y., "Trace-based microarchitecture-level diagnosis of permanent hardware faults," Proc. IEEE International Conference on Dependable Systems and Networks, DSN 2008.

[13] Carretero, J., Chaparro, P., Vera, X., Abella, J., Gonzalez, A., "Implementing End-to-End Register Data-Flow Continuous Self-Test," IEEE Transactions on Computers, pp. 1194-1206, Aug. 2011.

[14] Xenoulis, G., Gizopoulos, D., Psarakis, M., Paschalis, A., "Instruction-Based Online Periodic Self-Testing of Microprocessors with Floating-Point Units," IEEE Transactions on Dependable and Secure Computing, , vol. 6, no. 2, pp. 124-134, April-June 2009.

[15] Rodrigues, R., Kundu, S., Khan, O., "Shadow checker (SC): A low-cost hardware scheme for online detection of faults in small memory structures of a microprocessor," Proc. IEEE International Test Conference (ITC), pp.1-10, Nov. 2010.

[16] R. Baumann, "Soft Errors in Advanced Computer Systems, " IEEE Design & Test of Computers, vol. 22, no. 3, pp. 258-266, May-June 2005.

[17] www.itrs.net.

[18] DeOrio, A., Wagner, I., Bertacco, V., "Dacota: Post-silicon validation of the memory subsystem in multi-core designs," Proc. IEEE 15th International Symposium on High Performance Computer Architecture, HPCA 2009, pp. 405-416, Feb. 2009.

[19] http://www.redbooks.ibm.com/abstracts/sg244757.html.

[20] http://www1.ibm.com/support/docview.wss?uid=pos1R1003968&aid=1.

[21] Meisner, D., Gold, B., and T. F. Wenisch, "PowerNap: eliminating server idle power," Proceeding of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS '09), pp. 205-216.

[22] Deng, D., Lo, D., Malysa, G., Schneider, S., and G. E. Suh, "Flexible and Efficient Instruction-Grained Run-Time Monitoring Using On-Chip Reconfigurable Fabric," Proceedings of the 43rd IEEE/ACM International Symposium on Microarchitecture (MICRO '43), 2010, pp. 137-148.

[23] Lee R. B., "Multimedia extensions for general-purpose processors," Proc. SIPS 97 - IEEE Workshop on Signal Processing Systems, pp. 9-23, Nov 1997.

[24] Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M., "Single-ISA heterogeneous multi-core architectures: the potential for processor power reduction," Proc. 36th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-36, pp. 81- 92, Dec. 2003

[25] Li, C.; Ding, C., and Shen, K., "Quantifying the cost of context switch," Proceedings of the 2007 ACM workshop on Experimental computer science (ExpCS '07), Article 2 .

[26] Yilmaz, M., Tehranipoor, M., Chakrabarty, K., "A Metric to Target Small-Delay Defects in Industrial Circuits," IEEE Design & Test of Computers, vol. 28, no. 2, pp. 52-61, March-April 2011.

[27] Hu C.K. et al., "Scaling Effect on Electromigration in On- Chip CuWiring," Proc. International Electron Devices Meeting, 1999.

[28] Stathis, J.H., " Reliability Limits for the Gate Insulator in CMOS Technology," IBM Journal of R&D, Vol. 46, 2002.

[29] Ogawa, E.T., et al., "Leakage, Breakdown, and TDDB Characteristics of porous low-k silica based interconnect materials," Proc. International Reliability Physics Symposium, 2003

[30] Kim K-S., Mitra, S., Ryan, P.G., "Delay defect characteristics and testing strategies," IEEE Design & Test of Computers, vol. 20, no. 5, pp. 8- 16, Sept.-Oct. 2003.

[31] Sieworek D. P., and Swartz, R. S. (Eds.). Reliable Computer Systems: Design and Evaluation. A K Peters, 3rd edition, 1998.

[32] International Technology Roadmap for Semiconductors Report, www.sematech.org

[33] Kang K.; Gangwal, S.; Park, S.P.; Roy, K. , "NBTI induced performance degradation in logic and memory circuits: how effectively can we approach a reliability solution?," Proc. Design Automation Conference, ASPDAC 2008, pp.726-731, March 2008

[34] Benso, A,. Di Carlo, S., Natale, G., Prinetto, P., "A Watchdog Processor to Detect Data and Control Flow Errors," Proc. 9th IEEE On-Line Testing Symposium, p. 144, 2003

[35] Slegal, T.J et al., "IBM's S/390 G5 microprocessor design," IEEE Micro, March - April 1999.

[36] Smolens, J.C.; Gold, B.T.; Kim, J.; Falsafi, B.; Hoe, J.C.; Nowatryk, A.G., "Fingerprinting: bounding soft-error-detection latency and bandwidth," IEEE Micro, vol. 24, no.6, pp.22-29, Nov.-Dec. 2004