Architectures for Dependable Modern Microprocessors

Nikolaos Foutris[1]

[1] Department of Informatics & Telecommunications,
University of Athens, Athens, Greece,
nfoutris@di.uoa.gr

Abstract. Technology scaling, extreme chip integration and the compelling requirement to diminish the time-to-market window, has rendered microprocessors more prone to design bugs and hardware faults. Microprocessor validation is grouped into the following categories, based on where they intervene in a microprocessor's lifecycle: (a) silicon debug: the first hardware prototypes are exhaustively validated, (b) manufacturing testing: the final quality control during massive production, and (c) in-field verification: runtime error detection techniques to guarantee correct operation. The contributions of this thesis are the following: (1) Silicon debug: We propose the employment of deconfigurable microprocessor architectures along with a technique to generate self-checking random test programs to avoid the simulation step and triage the redundant debug sessions, (2) Manufacturing testing: We propose a self-test optimization strategy for multithreaded, multicore microprocessors to speedup test program execution time and enhance the fault coverage of hard errors; and (3) In-field verification: We measure the effect of permanent faults performance components. Then, we propose a set of low-cost mechanisms for the detection, diagnosis and performance recovery in the front-end speculative structures. This thesis introduces various novel methodologies to address the validation challenges posed throughout the life-cycle of a chip.

Keywords: Dependability, silicon debug, testing, error, bug

1 Introduction

The evolution of semiconductor technology and computer architecture has radically transformed our world throughout the last decades. However, the combination of technology scaling and extreme chip integration, along with the compelling requirement to diminish the time-to-market window, has rendered microprocessors more prone to design bugs and hardware faults. The goal of this thesis is to provide solutions to the validation challenges posed from the microprocessor products throughout the life-cycle of a chip.

Microprocessor validation is grouped into the following categories, based on where they intervene in a microprocessor's lifecycle: (a) silicon debug: the first hardware

[1] *Dissertation Advisor: D. Gizopoulos, Associate Professor*

prototypes are exhaustively validated, (b) manufacturing testing: the final quality control during massive production, and (c) in-field verification: runtime error detection techniques to guarantee correct operation. The contributions of this thesis are the following:

- **Silicon debug**: We propose the employment of deconfigurable microprocessor architectures along with a technique to generate self-checking random test programs to (a) avoid the time- and the resource-consuming simulation step, (b) triage the redundant debug sessions, and thus to accelerate silicon debug [2] [4].
- **Manufacturing testing**: We propose a self-test optimization strategy for multithreaded, multicore microprocessors to (a) speedup test program execution time, (b) enhance the fault coverage of hard errors, and thus to make manufacturing testing more efficient [1].
- **In-field verification**: We measure the effect of permanent faults performance components. Then, we propose a set of low-cost hardware-based mechanisms for the detection, diagnosis and performance recovery in the front-end speculative structures [5] [7] [10].

The share of silicon debug in the overall microprocessor chips development cycle is rapidly expanding. The validation step that detects the vast majority of design bugs is the one that stresses the silicon prototypes by applying huge numbers of random tests. Despite its bug detection capability, this step is constrained by the extreme computing needs for random test program simulation. Moreover, another major bottleneck and source of "noise" of this phase is that large numbers of random test programs fail due to the same or similar design bugs. This redundant behaviour adds long delays in the debug flow since each failing random program must be separately examined, although it does not usually bring new debug information. This thesis addresses both challenges of silicon debug. A self-checking methodology is proposed for generating random test programs (exploiting the ISA diversity property) that detect bugs by comparing the results of equivalent instructions combined with a technique to triage the failing test programs into categories with common failure modes. The proposed framework: (a) improves bug detection efficiency, (b) reduces the redundant debug session, and thus accelerates silicon debug.

When a sufficient level of coverage is reached the microprocessor design enters the production stage, where a last quality control is performed to detect any manufacturing defect. Functional self-testing forms an integral part of manufacturing test flow due to its at-speed testing and non-intrusive nature. Multithreaded (MT) SBST methodology proposes a novel self-test optimization strategy for multithreaded, multicore microprocessor architectures (OpenSPARC T1 microprocessor model). The proposed self-test program execution optimization aims to: (a) take maximum advantage of the available execution parallelism provided by multiple threads and multiple cores, (b) preserve the high fault coverage that single-thread execution provides for the processor components, and (c) enhance the fault coverage of the thread-specific control logic. MT-SBST methodology significantly speeds up self-test time, while at the same time it improves the overall fault coverage.

The combination of design complexity, shrinking time-to-market windows, and wear-out effects increases the failure probability of modern design and leads microprocessor manufactures to integrate numerous in-field verification mechanisms.

Trends such as low-voltage operation and process scaling are expected to significantly increase the rate of faults experienced by silicon. Their impact on a core's non-cache SRAM structures has not been accurately quantified. Faults in these structures will not affect correctness, but can cause severe performance degradation and variability among otherwise identical cores. We first classify and quantify the impact of permanent faults in the performance components of modern microprocessors. Then, we propose a low-cost microarchitectural mechanism that exploits the self-verification property of predictors to achieve performance recovery.

This thesis introduces various novel methodologies to address the validation challenges posed throughout the life-cycle of a chip. The proposed techniques make the validation process more efficient and are easily applicable to the existing industrial flow.

2 Silicon debug

Aggressive technology scaling and extreme chip integration, combined with the compelling requirement to diminish the time-to-market window have rendered microprocessors more prone to design bugs than ever. As a result, silicon debug – the process of validating and debugging a new microprocessor design on its first silicon prototype chips – has evolved to a critical, time-consuming, and labour-demanding step in a chip's development flow [11]. Recent trends [16] show that the time spent from the arrival of the first silicon prototype chip to high volume production ramping up is steadily growing, while the ratio between the size of the design and the debug teams has reached 2:1. Thus, an efficient silicon debug approach that promptly detects and eliminates the design bugs before volume production can make the difference between success and failure of a microprocessor product.

Silicon debug starts with the arrival of the first prototypes and often continues well after a product has gone to volume production. A comprehensive suite of test programs covering many test scenarios are executed on the prototype chips to detect bugs that can be anything from logic/functional bugs, electrical or process-related bugs to mask-related manufacturing defects [14]. Subsequently, for each failing test program (one that does not execute correctly due to a bug), separately, a systematic debug phase is performed by the debug engineers to identify the root cause of the failure.

Massive application of automatically generated random test programs on the prototype microprocessor chips is one of the most effective parts of silicon debug [13]. Despite its bug detection efficiency, this step is constrained by extreme computing needs for random tests simulation to extract the bug-free memory image for comparison with the actual silicon image. Another major bottleneck and source of "noise" in this phase is that large number of random test programs fail due to the same or similar design bugs. This redundant behaviour prolongs silicon debug phase since each failing random test program must be exclusively root-cause analysed, although it does not usually bring new debug information. Finally, volume production may be further prolonged due to bugs that lurk behind other bugs. These blocking

bugs stall the execution of the subsequent tests, since no workaround exists and therefore additional re-spins are needed.

This work introduces a silicon debug methodology for microprocessors with two major objectives: (a) increase coverage by applying more tests to silicon prototypes; and (b) reduce validation time by triaging the redundant failing random test programs. The methodology does so by exploiting (1) the inherent diversity of microprocessor instruction sets to eliminate the time consuming simulation step by employing self-checking tests; and (2) the property that allows hardware components to be deconfigured without compromising microprocessor's functional completeness to bucketing the redundant failing test programs. Figure 1 shows an overview of the flow.

A. Test generation: The fundamental first step is the identification of ISA diversities, i.e. microprocessor instruction equivalences, and the population of the ISA diversity database. The database contains for each instruction a list of equivalent instruction sequences. Then, the flow is fed with the random test programs (original RiTs) already generated (but not simulated) by sophisticated random test program generators that all microprocessor manufactures internally use [11] [12]. We pair each original RiT with an Equivalent RiT to generate an enhanced RiT. An eRiT is automatically generated from an original RiT replacing its instructions with their equivalent counterparts that have been stored in the ISA diversity database. Finally, a checking code compares the stored results of the original RiT and the eRiT to identify mismatches. A mismatch indicates a potential silicon bug.

B. Bug detection: Combining the self-checking method, with a hardware replay mechanism (Figure 2– right part) enables the extraction of as much as possible useful debugging information regarding the bug detection capability of each test program and provides a fast workaround solution to bypass blocking bugs. The hardware mechanism records the failing comparisons when mismatches are detected and replays the execution of the original RiT by replacing the execution of the offending instruction with its equivalent. In particular, the "replacement" is done on-the-fly using the program counter of the store instructions saved in buffers store-addr and estore-addr. During the first run of the enhanced RiT, the checking code finishes with the mismatches between the set of *k* responses of the original RiT and the eRiT stored in mids-queue (mismatch id queue), with mid between 0 and k. If mid = 0 (i.e. the queue is empty), then there is no mismatch and the chip passes the enhanced RiT; debug continues with the next RiT.

If queue is not empty (i.e mid > 0) the enhanced RiT will be replayed mid times, because store[mid] and estore[mid] instructions generated different results (Figure 3 – instr.16). The key functionality of the mechanism is that when a mismatch is detected between store[i] and estore[i], during replay, instead of executing the "buggy" code between store[i–1] and store[i], the processor executes the equivalent code between estore[i–1] and estore[i]. The mismatch has been bypassed, subsequent responses are not corrupted and if the remaining test can detect another mismatch (more bugs) it is allowed to do so. A list of mismatch identifiers (mids) is the log information our method provides. An integer m in the log (an entry in the mids-queue) means that: (a) the mth pair of stores produced a mismatch, i.e. store[m] and estore[m] produced different results; (b) the code between store[m–1] and store[m] has been replaced by the code between estore[m–1] and estore[m] and the original RiT continued. These

two pieces of information can help the debug engineer identify the offending instructions and work on them.

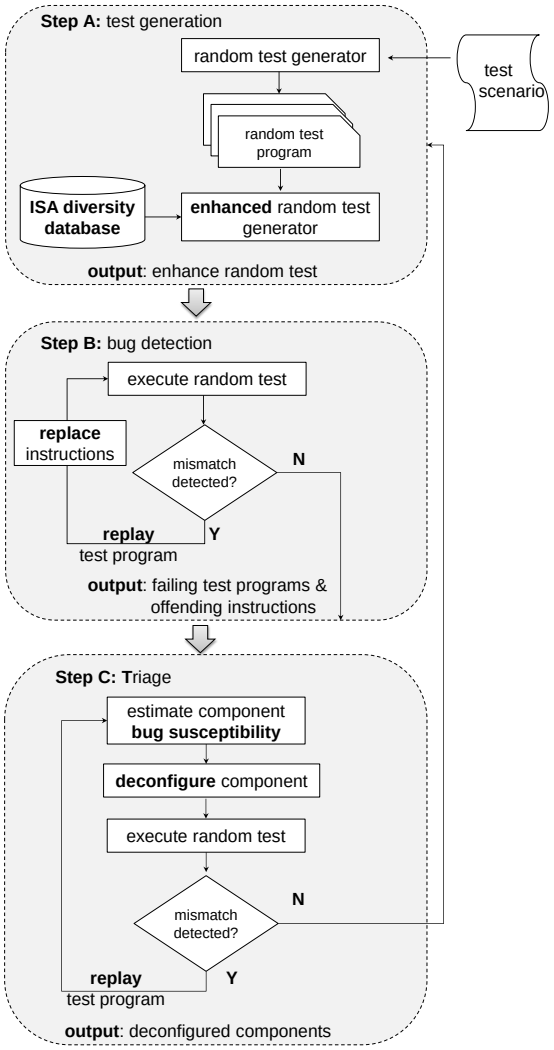

Figure 1: The proposed silicon debug flow.

C. Triage: As soon as bug detection phase finishes, hardware-assisted triage begins. Hardware triage is assisted through the integration of the triage mechanism (Figure 2– left part). For each failing self-checking random test, the triage mechanism selects the component that is most susceptible to contain a bug and deconfigures it in the next execution of the failing random test program. This process is repeated until the test program is correctly executed (i.e. the bug has been "masked" by the sequence of deconfigurations). All test programs that eventually execute correctly after the same sequence of deconfigurations are grouped into the same "bucket". Intuitively, the bug

that causes the failure most probably resides within the components that have been deconfigured before the test executes correctly.

The outcome of this step is a list of components that have been deconfigured and is stored in the component buffer (each entry of this array saves the id of the component). The interpretation of the list provides the following triage-related information: (a) Empty list. The random test program was correctly executed. No failure detected; no debug action required in the morning, (b) List contains a set of the deconfigurable components. The random test program was correctly executed after components {Ck, Cn, Cm, Cq} have been deconfigured. The list of components indicates a "bucket" of failing test programs. All test programs ending with the same list of deconfigurations are grouped together; and (c) List contains all deconfigurable components. The random test program fails even after all deconfigurable components are turned off. No triage grouping information; the random test must be separately debugged.

At the end of the multiple hardware-enabled test program re-executions, the contents of the component buffer and the mismatch identifier queue (grey colored boxes in Figure 2) are downloaded along with the remaining memory image of the prototype on the host machine (i.e. dedicated server that controls the entire validation campaign) for further analysis by the debug engineers. It should be noted that the proposed methodology detects bugs (both logical and electrical) with the following characteristics: (i) their excitation does not depend on the operational conditions (temperature, voltage, frequency); and (ii) they continue to manifest themselves despite the deconfiguration of components from the overall design.

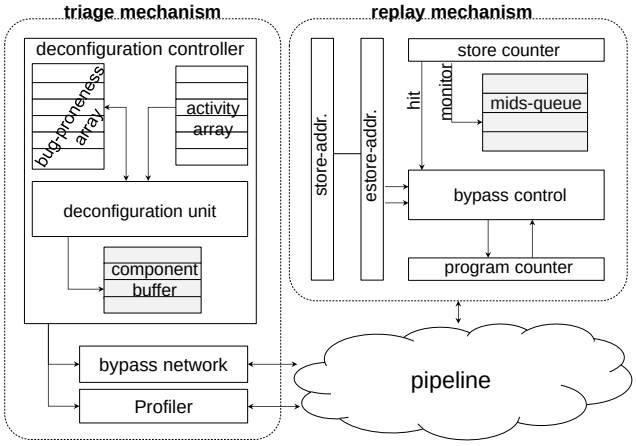

Figure 2: The proposed hardware mechanism for silicon debug acceleration.

To evaluation the proposed silicon debug methodology, we set up the tool chain on top of the PTLsim [18] architectural simulator as presented in [4].

First set of experiments: We compare our methodology, in terms of bug efficiency, with the traditional flow (mismatches are only detected off-line comparing the memory dumps of the actual execution with the expected memory dump contents from simulation) and with two other self-checking validation approaches [15] [17]. For each of the three methods, we use the same original RiT (4K instructions) as input and we enhance it according to the basic idea of each method. Our methodology

detects all 1K bugs injected into the simulator (Figure 3) because we stopped generation of more RiTs when all the injected bugs were detected. The traditional flow detects 928 bugs (coverage 90.54%). This difference, against the proposed method, is explained by the activation of more hardware areas by the equivalent RiT. The approach of [17] detects 903 bugs (coverage 88.10%) because there are cases where an instruction cannot be reversed. Furthermore, the flexibility of the ISA diversity concept to deploy equivalent instructions which activate totally different paths in processor's logic provides us with the ability to avoid bug masking conditions. Finally, [15] detects 210 bugs (coverage 20.49%) because it can only detect electrical bugs, since a logic bug will act in an identical way in both original and duplicated instruction. For a complete silicon debug plan (trillions of instructions), we expect our approach to have the same bug efficiency as the traditional flow since our bug detection capability relies on the original RiTs which are carefully generated by sophisticated industrial random generators. The advantage of our method is that by avoiding the time-consuming simulation step it is able to apply many more RiTs and thus detect potential bugs much earlier.

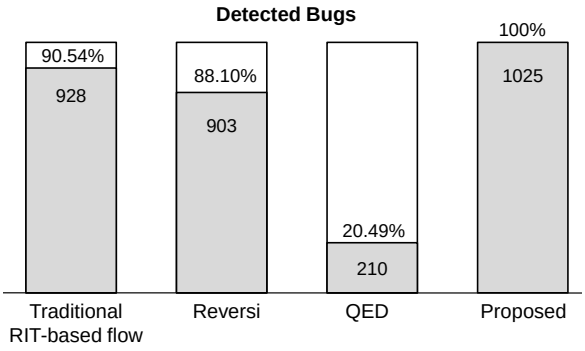

Figure 3: Design bug coverage for the four different methods.

Second set of experiments: The proposed method refines the debug information using the hardware replay mechanism. During our bug injection experiments, we observed that the average number of different bugs that were detected by a single RiT is about 4. To verify the effectiveness of our approach on refining debug information, we activate the hardware replay mechanism in our infrastructure (the triage mechanism is disabled) and conducted a second set of experiments: we injected all the bugs at the beginning of the simulation and executed all RiTs with the highest bug detection capability. The proposed hardware mechanism detected all the injected bugs (through bypassing the offending instructions with their equivalents in the replay executions). This is a significant benefit of the proposed framework compared to the traditional flow which requires more tests to detect the same number of bugs.

Third set of experiments: To demonstrate the benefits of the triage mechanism on test program triaging, we have selected a set of 10 hard-to-detect logic bugs from the set of injected bugs distributed among the deconfigurable modules of PTLsim simulator (we characterize them as hard-to-detect because all 10 bugs are detected by a small number of test programs; smaller than the average case). Furthermore, all 10 design bugs are together injected from the beginning of the bug injection campaign,

as an attempt to model more accurately the silicon debug environment where all bugs can co-exist in the prototype chip. We repeated the experiments only for a subset of the initial random test programs that are affected from them; these are 341 test programs.

Table 1 presents details about the selected design bugs. The first column is the id of each bug, while the second column gives the microprocessor component in which the bug resides. Issue $Queue_1$ and Issue $Queue_2$ refer to different components in the microprocessor design (Issue $Queue_1$ for the integer cluster, and Issue $Queue_2$ for the floating point cluster). The third column shows the number of test programs affected by each design bug when injected individually (from the first set of experiments) and the last column provides a short description.

Bug ID	Component	Failing Test Programs	Bug Description
1	Conditional Predictor	45	Update fetch address on branch misprediction fails
2	RAS	10	Incorrect push to stack
3	Issue $Queue_1$	32	Dependent uop issued, while producer is waiting in ready-to-write-back state
4	Issue $Queue_2$	21	Entry not flushed on a branch misprediction
5	Floating Point Unit	50	Incorrect rounding operation
6	Data cache	17	Valid array logic; invalid data read
7	Load Queue	47	Load to store aliasing
8	Store Queue	29	Store data before address gets valid
9	Reorder Buffer	48	Commit entry more than once
10	Reorder Buffer	42	Invalid control bit activation
Total		**341**	**-**

Table 1: Details of 10 hard-to-detect design bugs.

Figure 4 shows the results for this set of experiments. The horizontal axis presents the different "buckets" of failing random test programs that are formed when the proposed methodology is applied. The vertical axis shows the number of failing test programs of each bucket.

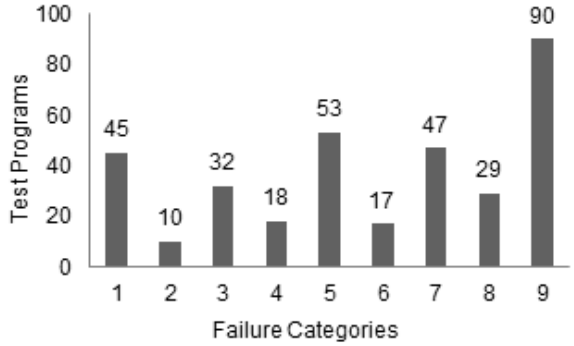

Figure 4: Failure categories for the 341 failing test programs.

The application of the proposed methodology with the deconfiguration mechanisms enabled results in a triaging of the 341 random test programs in 9 different failure categories shown in Figure 4:

- Failure categories 1, 2, 3, 4, 6, 7, and 8 group the test programs that are affected exclusively from the design bugs in one of the following microprocessor components: Conditional Predictor, RAS, Issue $Queue_1$, Issue $Queue_2$, Data Cache, Load and Store Queues, respectively. As a result, when the deconfiguration controller turned the corresponding microprocessor component off, the bug is "masked" and the test program execution is correct.
- Failure category 5 groups 53 random test programs, while the expected number of test programs affected from a design bug in the FPU unit is 50. The reason for that is that these particular test programs (3 from Issue Queue2) were able to detect more than one design bugs (design bugs injected both in the Issue Queue2 and the FPU). As a result, only when both buggy microprocessor components were deconfigured the re-execution of the test program results in a correct execution.
- Failure category 9 includes the test programs that fail due to bugs 9 and 10 injected in the Reorder Buffer's logic. The deconfiguration mechanisms were unable to distinguish these design bugs into different categories, since both of them were inside the deconfiguration granularity of the ROB structure. Specifically, these bugs reside in neighboring entries of the re-order buffer and manifest themselves as invalid dependency re-dispatching when a mis-speculation happens. Therefore, the same sequence of deconfiguration results in masking both bugs..

Clearly, the proposed flow has a profound impact on the effectiveness of silicon debug and greatly accelerates root cause analysis by removing the "noise" of redundant random tests that fail due to the same underlying bug (the 341 initial debug sessions are reduced to only 9 in the last set of experiments).

3 Conclusions

Today, the pervasiveness of microprocessors, the most complex and immensely powerful application of electronics, in our society goes far beyond the wildest imagination. The same path that is leading technologies toward these remarkable achievements is also making them increasingly unreliable posing a threat to our society. Silicon technology process scaling trends, modern architecture complexity and the compelling requirement to diminish the Time-to-Market threaten to create a "validation wall". As a result, semiconductor industry and academic researchers must explore radical solution and develop innovative techniques to address the dependability challenges of the current and the forthcoming microprocessors. This thesis introduced novel methodologies to address the validation challenges posed throughout the life-cycle of a microprocessor.

Microprocessor validation is grouped into three categories, based on where they intervene in a microprocessor's lifecycle: (a) silicon debug: the first hardware prototypes are exhaustively validated, (b) manufacturing testing: the final quality control during massive production, and (c) in-field verification: runtime error detection techniques to guarantee correct operation. This thesis introduces various

techniques to tackle the challenges of microprocessor validation targeting to: (a) make the dependability process more efficient; and (b) be easily applicable to the existing industrial flow. The contributions of this thesis are as follows:

- Silicon debug: The share of silicon debug in the overall microprocessor chips development cycle is rapidly expanding due to the ever growing design complexity and the limited throughput of pre-silicon verification methods. Massive application of short random test programs on the prototype microprocessor chips is one of the most effective parts of silicon debug. Despite its bug detection capability, it is constrained by extreme computing needs for random test programs simulation to extract the bug-free memory image. Another major bottleneck and source of "noise" in this phase is that large numbers of random test programs fail due to the same or similar design bugs. This redundant behavior adds long delays in the debug flow since each failing random test program must be separately examined, although it does not usually bring new debug information. We proposed the employment of self-checking random test programs along with a deconfigurable microprocessor architecture to avoid the time-consuming simulation step, triage the redundant debug sessions and thus accelerate silicon debug. To do so, we exploited the inherent diversity found in all popular Instruction Set Architectures (ISAs) and the ability to deconfigure hardware modules without affecting the functional completeness of a design. Detailed evaluation of the method on an x86 microprocessor model demonstrated its effectiveness in accelerating silicon debug.
- Manufacturing testing: We presented an efficient multithreaded (MT) SBST methodology that optimizes self-test time taking maximum advantage of thread-level parallelism while at the same time enhances the self-test program error detection capability on the thread-specific control logic of the processor. The methodology contributed to the effective application of SBST in manufacturing testing. Our experiments on OpenSPARC T1 revealed that the proposed methodology improved significant test execution time at both the core level (3.6 times) and the processor level (6.0 times) against single-threaded execution, while at the same time it improves fault coverage. Compared with a straightforward multithreading approach, it reduces the self-test time at both the core level and the processor level by 33% and 20%, respectively. Overall, our methodology guarantees high stuck-at fault coverage (88% for the entire processor, more than 1.5M logic gates), which is the highest coverage ever reported in the literature by a software-based functional test methodology in such a complex industrial microprocessor.
- In-filed verification: Aggressive technology scaling along with low voltage operation exacerbates the likelihood and rate of hard faults not only in large SRAM arrays (such as cache memories), but also in non-SRAM microprocessor structures. Some of the largest non-cache SRAM structures support speculation such as the branch predictor tables, the branch target buffers, and the data prefetcher. Faults in these structures will not affect correctness, but can cause severe performance degradation and variability among otherwise identical cores. We accurately classified and quantified the performance impact of hard faults in non-SRAM structures over a set of CPU benchmarks. To do so, we applied a statistically safe fault injection campaign for single and multiple faults a modified version of the cycle-accurate x86 architectural simulator PTLsim running the SPEC CPU2006 suite. Our evaluation revealed significant differences in the effect of faults and their performance impacts

across the components as well as within each component. In particular, we demonstrated that a very large fraction (44% to 96%) of hard faults in these components leads to performance fluctuation, Furthermore, faults in the data prefetcher degrade IPC by up to 26%, compared to fault-free operation, while faults on the branch prediction unit reduce IPC by more than 16%, respectively. Moreover, we found that faults in these components can substantially increase the performance variability across identical cores. Finally, we proposed low-cost microarchitectural techniques to diagnose predictor faults and recover the performance loss. Our techniques exploited the self-verification property of predictors to achieve performance recovery at lower cost than comparable techniques. We found that our solutions can recover almost all performance loss and virtually eliminate performance variability among cores.

The research outcomes of this thesis open the door to several future directions. Future systems architectures must be designed to facilitate hardware validation. In particular, future solutions should have adhered to the following guideline principles: (a) low-power, (b) negligible area overhead, (c) scale with design complexity; and (d) highly automated. In the silicon debug domain, future research should focus on the automation and standardization of the design bug detection and root-cause analysis process. Furthermore, this thesis demonstrated the effectiveness of software-based techniques in accelerating manufacturing testing and guaranteeing a high level of fault coverage. This may be an indication that future microprocessors should devote valuable silicon estate in hardware hooks that enable the at-speed, low-cost testing. The growing demand for high-performance computer systems pushes computer architects to integrate numerous performance mechanisms in the microprocessor designs. However, functional correctness is prioritized over performance correctness. This work revealed that faults in performance components can lead to noticeable performance loss and variability in otherwise identical cores. Therefore, future designs must integrate mechanisms to continuously monitor the system performance health and applying contingency actions. Finally, a vital future research direction is to bridge the gap between silicon debug, manufacturing testing and in-field verification techniques through the development of cross-cutting solution that will operate throughout the entire life-cycle of a microprocessor.

The vital challenge of future technologies is to build dependable systems. This thesis proposed various novel techniques to make the validation process, throughout microprocessor life-cycle, more effective in terms of bug/error detection efficiency, resource- and time-budget. We hope that the contributions presented in this thesis will advance the research in manufacturing dependable microprocessor architectures and will find applicability in future commercial microprocessor products.

References

[1] N.Foutris, M.Psarakis, D.Gizopoulos, A.Apostolakis, X.Vera and A.Gonzalez, MT-SBST: Self-Test Optimization in Multithreaded Multicore Architectures, In IEEE International Test Conference (ITC), 2010.

[2] N.Foutris, D.Gizopoulos, M.Psarakis, X.Vera and A.Gonzalez, Accelerating Microprocessor Silicon Validation by Exposing ISA Diversity. In ACM/IEEE International Symposium on Microarchitecture (MICRO), 2011.
[3] T.Ramirez, E.Herrero, N.Axelos, J.Carratero, N.Foutris, D.Sanchez, X.Vera, Mitigating Lower Layer Failures with Adaptive System Reconfiguration, International Symposium on Mixed Design of Integrated Circuits and Systems (MIXDES), 2012.
[4] N.Foutris, D.Gizopoulos, X.Vera and A.Gonzales, Deconfigurable microprocessor architectures for silicon debug acceleration. In ACM/IEEE International Symposium on Computer Architecture (ISCA), 2013.
[5] N.Foutris, D.Gizopoulos, J.Kalamatianos and V.Sridharan, Assessing the Impact of Hard Faults in Performance Components of Modern Microprocessors, In IEEE International Conference on Computer Design (ICCD), 2013.
[6] M.Kaliorakis, N.Foutris, D.Gizopoulos and M.Psarakis, Online error detection in multiprocessor chips: A test scheduling study, IEEE International On-Line Testing Symposium (IOLTS), 2013.
[7] N.Foutris, D.Gizopoulos, J.Kalamatianos and V.Sridharan, Measuring the Performance Impact of Permanent Faults in Modern Microprocessor Architectures, IEEE International On-line Testing Symposium (IOLTS), 2013.
[8] M.Kaliorakis, M.Psarakis, N.Foutris and D.Gizopoulos, Parallelizing Online Error Detection in Many-core Microprocessor Architectures, Joint Euro-TM/Median Workshop on Dependable Multicore and Transactional Memory Systems (DMTM), 2014.
[9] M.Kaliorakis, M.Psarakis, N.Foutris and D.Gizopoulos, Accelerated Online Error Detection in Many-core Microprocessor Architectures, In IEEE International VLSI Test Symposium (VTS), 2014.
[10] N.Foutris, D.Gizopoulos, A.Chatzidimitriou, J.Kalamatianos and V.Sridharan, Performance Assessment of Data Prefetchers in High Error Rate Technologies, In IEEE Silicon Errors in Logic – System Effect (SELSE), 2014.
[11] M.Abramovici, P.Bradley, K.Dwarakanath, P.Levin, G.Memmi, D.Miller. "A reconfigurable Design-for-Debug Infrastructure for SoCs", In ACM/IEEE Design Automation Conference (DAC), 2006.
[12] M.Behm, J.Ludden, Y.Lichtenstein, M.Rimon and M.Vinov, "Industrial Experience with Test Generation Languages for Processor Verification", In ACM/IEEE Design Automation Conference (DAC), 2004.
[13] T.Bojan, F.Igor and M.Robert, Intel's Post Silicon Functional Validation Approach, IEEE High Level Design Validation and Test Workshop (HLDVT), 2007.
[14] International Technology Roadmap for Semiconductors, 2009.
[15] T.Hong, Y.Li, S-B.Park, D.Mui, D.Lin, Z.A.Kaleq, N.Hakim, H.Naeimi, D.S.Gardner and S.Mitra, "QED: Quick Error Detection Tests for Effective Post-silicon Validation, In IEEE International Test Conference, 2010.
[16] Y-C.Hsu, F.Tsai, W.Jong and Y-T Chang, "Visibility Enhancement for Silicon Debug", In ACM/IEEE Design Automation Conference (DAC), 2006.
[17] I.Wagner and V.Bertacco, "Reversi: Post-silicon Validation System for Modern Microprocessors", In IEEE International Conference on Computer Design (ICCD), 2008.
[18] M.Yourst, "PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator", In IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2007.