Hardware Performance Counters for System Reliability Monitoring

Elena Woo Lai Leng, Mark Zwolinski and Basel Halak
Department of Electronics and Computer Science
University of Southampton, Southampton, United Kingdom SO17 1BJ
Email: L.L.Woo@soton.ac.uk

***Abstract*—As technology scaling reaches nanometre scales, the error rate due to variations in temperature and voltage, single event effects and component degradation increases, making components less reliable. In order to ensure a system continues to function correctly while facing known reliability issues, it is imperative that the system should have the means to detect the occurrence of errors due to the presence of faults. A system that behaves normally (no error detected in the system) exhibits a profile, and any deviations from this profile indicate that there is an anomaly in the system. In this paper, we propose to use hardware performance counters (HPCs) to measure events that occur during the execution of the program. We explore the various counters available which could be use to identify the anomalous behaviour in the system and develop a methodology to observe the anomalies using HPCs by creating a fault-free pattern and observing any subsequent changes in that pattern. We evaluate the proposed technique using GemFI, an architectural simulator based on Gem5 with additional fault injection capabilities. We compare the results obtained at the end of the execution with data collected during a time interval. Our results show that HPCs can be used to identify anomalous behaviour in a system that would lead to failure.**

I. Introduction

The decrease in transistor size and increase in integrated circuit performance driven by Moore's Law has enabled the growth of computing systems across various sectors and applications such as the aerospace, transportation, medical, super computing and commercial industries. While technology scaling has enabled computing systems to be built with better performance and higher transistor density per die at lower cost and power consumption, it has also resulted in decreasing reliability [1]–[3]. Unreliable transistors have a huge impact on computing systems, particularly on embedded systems that have limitations in terms of hardware resources, processor speed, power consumption and memory size [4].

It is known that reliability issues can cause anomalous behaviour in a system [5], [6] and that anomalous behaviour can lead to failures in the correct execution of applications. When a computer system is in operation, it is exposed to variations in power, performance and operating conditions [7], which increase the possibility of system-level anomalies. Online error detection can detect violations of system specifications at run-time, to try to ensure that less-than-perfect chips can still operate effectively even in the presence of faults.

There are several techniques that have been explored for online error detection. For example, Hari et al, [8], developed the Multicore SoftWare Anomaly Treatment (mSWAT) for anomaly detection. These low-cost hardware and software detectors, which are used to monitor and detect anomalous behaviour, have proven to be successful in detecting *fatal traps*, *hangs*, *high OS* and *panic*. However, there is an area overhead involved in these monitors being specifically built and added into the system.

Software Built-In-Self-Test (BIST) is another form of online error detection. While it is non-intrusive and does not require any change in the hardware design, as it utilises existing processor resources and instructions to perform self-testing, [9], it is only able to detect permanent faults [10]. There is also an issue of performance loss when the test program is executed on multiple cores; one solution to overcome performance loss is to run the test program in parallel, [11], however, the normal workload has to be suspended in all cores.

Another technique for online error detection is a redundancy-based technique where two or more independent threads execute the same program and its results are compared against each other. Dual modular redundancy, triple modular redundancy and N-modular redundancy are some of the well-known hardware redundancy techniques studied and applied but they have high overheads. Software redundancy techniques such as EDDI and SWIFT, [12], [13], focus on duplicating all instructions but this causes 100% overhead in performance. A recent approach, [14], uses a data-flow graph in which the redundant instance is placed in the graph and the output from the redundant instance is compared against that of the original instance. This technique however, is not very suitable for applications that intensively use the memory bus, as it shows significant performance degradation; for some benchmarks up to 23% overhead is recorded.

In this paper, we propose using *Hardware Performance Counters* (HPCs) to extract the features of a system for reliability monitoring. HPCs are sets of special-purpose counters built into processors to record events precisely and accurately as they occur. These counters are part of a special, dedicated unit in the central processing unit (CPU) called the Performance Monitoring Unit (PMU). This has the ability to access detailed information regarding the processor's functional units and caches, as well as memory, but the types of counters and the meaning of those counters varies from one processor to another processor due to architectural differences. As these hardware counters are built-in, there are no additional

overheads. The counters are incremented on an instruction-by-instruction basis, thus ensuring accurate results, [15], [16].

We have found no published papers to date on utilising hardware counters for error monitoring at system level for reliability. A system that behaves normally (functioning without any error) exhibits a certain pattern, thus any behaviour that deviates from that normal pattern should be identifiable. Our proposed method does not incur any additional overheads while the counter values are collected during program execution. Overall, the main contributions of our work are as follows:

- We explore the various counters available and hence decide the number of committed instructions, the number of function calls, the number of integer instructions and the number of load instructions that could be use to identify the occurrence of errors in a system;
- We develop a methodology to observe the anomalous behaviour using HPCs to create a fault-free pattern and to observe any change to the pattern; and
- Our results show that HPCs can be used to identify anomalous behaviour in a system that would then lead to failure.

This paper is organised as follows. Section II looks at how HPCs are applied in other applications. Our proposed methodology and experiment are presented in Section III. In Section IV, we discuss the data we obtained from our experiment. Finally, in Section V, we conclude the paper and present our future work.

II. Applications of HPCs in Other Areas

HPCs are mainly used to detect malicious activities or used for performance evaluation. For example, in [17], ConFirm is a low-cost technique that leverages the use of HPCs to detect malicious modification of firmware in embedded control systems by monitoring the computational paths in a subroutine of monitored firmware. It uses HPC as a signature to verify the execution of the computational paths.

In [18], BRAIN is a host based Distributed Denial-of-Service (DDoS) detection framework that uses HPCs to detect attacks on the application. This method uses a set of HPC statistics from network activities and correlates it with a set of HPC statistics from application activities to differentiate the behaviour of the host during load and attack. It is shown that correlating the HPCs between application and network can successfully detect DDoS with high accuracy and low cost and performance overheads.

Wang et al, [19], use HPCs to monitor and quantify the interference between virtual machines located in the same host and competing for shared physical resources. Using Last Level Cache (LLC) miss-rates, one of the many counters available, the data is fed into the interference prediction model to predict performance degradation between virtual machines and the information gathered can determine which virtual machine is utilising most of the resources.

Is another example of how HPCs are used for performance evaluation, Rasoolzadeh et al, [20] proposed to monitor L1 cache activity counters in order to estimate the workload and setting the Dynamic Voltage and Frequency Scaling (DVFS) based on the estimated workload. This method resulted in energy saving of 23% compared to the *on-demand* frequency setting policy used in Linux.

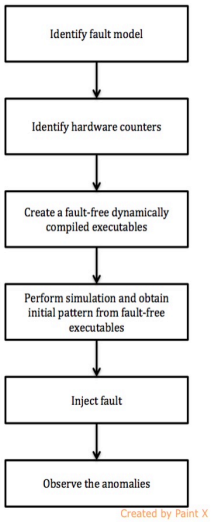


Fig. 1. The methodology set-out for this experiment

III. Methodology and Experiment

Similar to what was done elsewhere, [15], [18], we use HPCs to create fault-free execution profiles for several different type of benchmarks. Figure 1 gives a general overview of our methodology.

There are several different fault models used in digital circuits. In our experiment, we focus on the single stuck-fault (SSF) model because this fault model is applicable to many different physical fault regardless of whatever technology is applied.

We have chosen to use the following counters to profile the executions: the number of committed instructions (instructions that were executed); number of function calls; number of integer instructions; and number of load instructions.

Once we have identified the fault model and specific hardware counters for monitoring, we create fault-free executables from various benchmarks and perform the simulation using Gem5 and GemFI, described below. We obtain initial execution profiles using the counters. The next step is to inject fault and observe the anomalies recorded using the counters.

This experiment was conducted on a Microsoft Azure virtual machine, [21]. We created a Linux virtual machine with 16 central processing units (CPUs), 32 GBs of memory and 1TBs of data storage. We used Ubuntu version 14.04 for compatibility with Gem5 and GemFI.

A. Architectural Simulator

Gem5 [22] simulator is an instruction set simulator, widely used in computer architecture research. It supports various Instruction Set Architectures (ISAs) such as X86, ARM,

Alpha, Sparc, Mips and Power. Gem5 can operate in two modes: *System Call Emulation (SE)* and *Full System (FS)*. SE mode allows users to emulate most common system calls, thus avoiding the need to model devices or even an operating system (OS). In FS mode, Gem5 models complete system including the OS and devices, executing both user-level and kernel-level instructions.

GemFI [23], is a cycle accurate fault injection tool developed based on Gem5 with the primary objective of enabling fault injection. GemFI supports the Alpha and Intel X86 ISAs. There are two intrinsic functions provided by GemFI API:

- **void FI_init()** initialises the fault injection module.
- **void FI_activate (int id, int command)** is a pseudo-assembly instruction to toggle a fault on a specific thread. The thread is given a numerical identification number.

A set of faults is generated using the fault generator that comes together with GemFI. Each fault contains four attributes: *Location*; *Thread*; *Time*; and *Behaviour*.

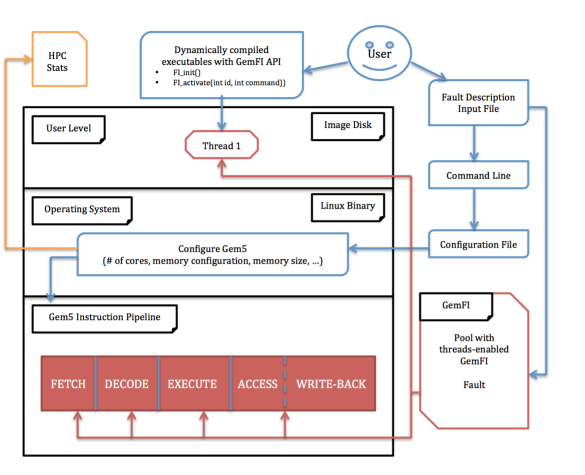


Fig. 2. Overview of the GemFI API, after [23]. The red components show possible fault injection locations; the red octagon is where the executables run.

B. Benchmarks

The benchmarks used in this experiment are from MiBench [24], which is a set of 35 embedded applications divided into six suites with each suite targeting a specific area of the embedded market. We have chosen the *basicmath*, *bitcount* and *qsort* benchmarks from the Automotive and Industrial Control suite, as well as *Dijkstra* from the Network suite. The basicmath program performs simple mathematical functions. The bitcount algorithm tests the bit manipulation ability of a processor by counting the number of bits in an array of integers and *qsort* uses the popular qsort algorithm to sort a large array of strings into ascending order. *Dijkstra* is a benchmark that calculates the shortest path between every pair of nodes in a graph.

C. Experimental Setup

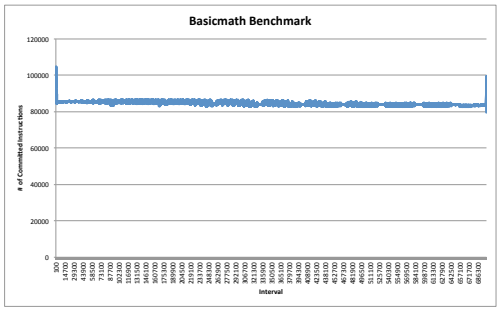
To extract the HPCs features that will be used to monitor system reliability, there are several steps:

1) Set up the benchmarks required for testing.
   Each benchmark was compiled dynamically in two versions – one in the original form and another with GemFI intrinsic functions added. Both versions were compiled for the X86 ISA. For *basicmath*, no input data was required, whereas for *bitcount*, the input data is an array of integers and for *qsort*, the input data contains a list of words. The input data for *Dijkstra* is a large graph in the form of an adjacency matrix. The executable files are then placed in the disk image serving as the virtual disk for GemFI.

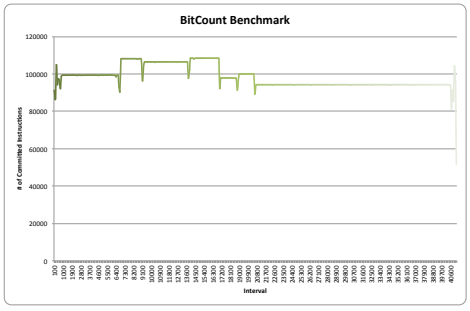
2) Perform the simulation.
   Simulation of the benchmarks was performed in both the Gem5 and GemFI simulators in FS mode. FS mode simulates the execution of the benchmarks in an OS-based simulation environment. A script file is created to assist in the execution of the benchmarks. After fault injection has been initialised and enabled, a set of faults is then created using the fault generator in GemFI. A fault configuration file describing the fault to be injected is provided for GemFI. This file is parsed at startup and each fault is injected into one of the four internal queues, which correspond to a pipeline stage. The simulation continues as normal until it is time for the fault to be injected. Figure 2 provides a general overview of how the simulation works using GemFI API. The blue lines indicates that the tasks belong to the user, the red lines indicate the responsibility of GemFI, and the orange line denotes the HPC values as outputs from the OS.

   Each experiment is executed six times: (i) initial run; (ii) with fault activation; (iii) fault injected in the Fetch pipeline; (iv) fault injected in the Decode pipeline; (v) fault injected in the Execute pipeline; and (vi) fault injected in the Load/Store pipeline. The fault model applied in this experiment is a stuck-at-1 fault model, and it is applied at every level in the pipeline. For each experiment conducted, the HPCs are traced using the method outlined in section III.C.3.

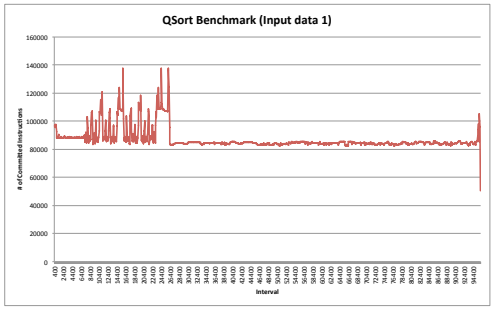
3) Trace and record the required HPC values.
   Two different tracing methods were tried to log the HPCs values obtained. The first method was to obtain the HPCs after the operating system (OS) has booted and another set of HPCs at the end of the execution of the benchmark. However, this method can only provide an indication that an error has occurred which causes the program to either hang, crash or provide incorrect output, but is unable to determine when the fault occurred. The second method was to log the HPCs values at certain intervals. Using this method, we can demonstrate that we are able to create an execution profile for each benchmark, and to detect the instance



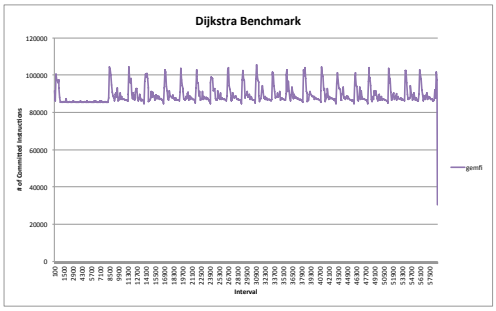
(a)



(b)



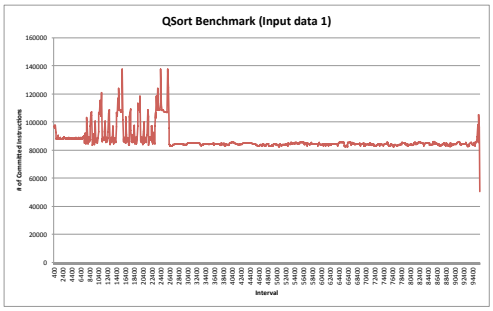
(c)



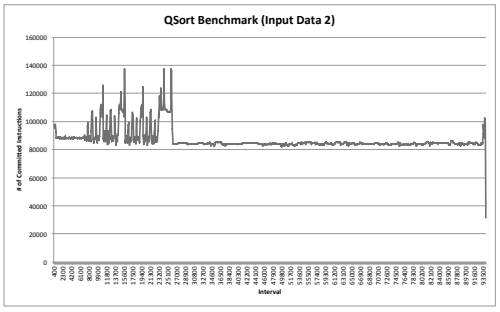
(d)

Fig. 3. Execution profiles based on the number of committed instructions for different benchmarks - (a) Basicmath, (b) Bitcount, (c) QSort and (d) Dijkstra

when an error has occurred. We found that tracing the HPC data at time intervals of 10ms (which is equivalent to 200,000 cycles) is sufficient to create a profile for each benchmark. The HPC data presented in section IV are only for the benchmarks and do not include the OS.



(a)



(b)

Fig. 4. Execution profiles of QSort benchmark - (a) Set input 1 and (b) Set input 2

IV. Discussion and Analysis

Figure 3 compares the execution profiles obtained from four different benchmarks using the number of committed instructions (axis Y) plotted against the time interval (axis X). By inspection, the profiles for each benchmark differ from one another, which suggests that HPCs can be used to identify the normal behaviour of the system. Profiles using the number of integer instructions, the number of function calls as well as the number of load instructions were plotted as well (but not displayed here due to space) and these profiles also show distinct differences, suggesting that it is sufficient to monitor the reliability of the system based on one or two counters.

We also tried to compare execution profiles for one benchmark but with two different sets of input data. We executed the QSort Benchmark with one set of input data that consists only of words and integers and a second set of input data that consists of words, integers and floating-point numbers. Figure 4 shows that the execution profiles generated with the two different sets of inputs still bear strong similarities to one another although the total number of committed instructions executed differs. This finding suggests that for this kind of fault model, regardless of any input used, the execution profile remains similar, and thus it is still possible to observe anomalies that may occur based on the profiles generated by the counter.

In our experiments, as we injected a stuck-at-1 fault in every pipeline, we discovered that this stuck-at-1 (or 0) fault led to errors such as *segmentation faults*, *invalid opcodes*, *kernel panics* and *invalid instruction pointers*. These errors

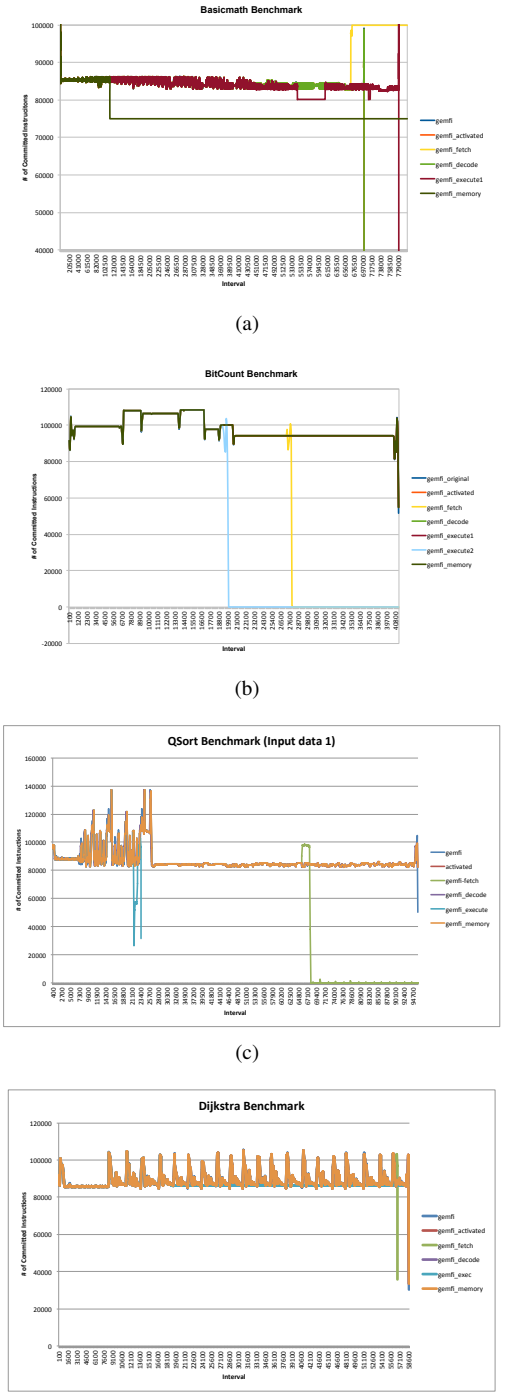
TABLE I
TOTAL INSTRUCTIONS FOR EACH BENCHMARK

Total Instructions	Benchmarks				
	Basicmath	Bitcount	QSort	QSort (2)	Dijkstra
GemFi	590984224	40508678	83324366	82063568	52370245
GemFI w/ Fault Activated	590989240	40511222	83339395	82065794	52373445
Injected Fault - Fetch	13718386802	27945500	59913002	421330792	50701576
Injected Fault - Decode	590989240	40511222	83244369	82065794	52373445
Injected Fault - Execute	586196426	20397641	21025988	42442442	302299133
Injected Fault - Memory	1410304814	40511222	83339395	82065792	52373446

will then cause the program to either *crash* or *hang*. Table I shows the total instructions executed for each benchmark. In particular, "GemFI" and "GemFI w/ Fault Activated" represent the baseline data for the program being executed successfully. The value in "GemFI w/ Fault Activated" will always be slightly higher compared to "GemFI" due to the additional intrinsic functions that are added. Table I also displays the total instructions for *QSort* and *QSort2* with two different sets of input data.

Table I list the total instructions executed after a fault is injected in every pipeline. Values that differ from the baseline "GemFI w/ Fault Activated" number indicate that an anomaly has occurred in the program. A value that is below the baseline indicates that the program terminates early, whereas a value that exceeds the baseline indicates that the program hangs. As we discussed earlier, obtaining the data at the end of the execution is only able to tell us whether the program has terminated successfully but cannot determine when an error has occurred. For example, consider the total instructions recorded for the *basicmath* benchmark when a fault is injected in the Execute pipeline. If we compare the value of 586196426 against the baseline 590989240, the initial conclusion would be the program had not terminated successfully. However, if we perform instructions tracing in a time interval, we can see that an error has occurred but that the program did terminate successfully. This is illustrated better in Figure 5 (a) where the profile shown by the maroon line indicates a drop at interval 553500 to value 80000 compared to the expected value between 84000 and 85000. The value 80000 stays for a number of cycles until the program managed to recover from the fault and resumed the execution at interval 615000 until completion. We have shown that by tracing the data at intervals and creating a profile using the HPCs, it is possible to capture even a slight change in the program.

In Figure 5 (a) for the Basicmath benchmark, the profile was able to capture the error occurring in the *fetch* pipeline (yellow line) and the *memory* pipeline (dark green line). As noted, it is also possible to detect an error that occurs due to a fault, but is recovered, as shown by the maroon line in the same figure. This figure shows how an error can be visible if we performed data tracing. Other benchmarks, Figure 5 (b), (c) and (d), also show that errors are detectable using the counters. In 5 (b), the HPCs were able to detect a system hang in the *fetch* pipeline (yellow line) and *execute* pipeline (blue line), whereas in 5 (c), the *execute* pipeline (blue line) shows that the program has crashed (hence the line stopped in the midst of execution), and the *fetch* pipeline (green line) shows the program hangs.



(a) (b) (c) (d)

Fig. 5. Execution profiles that shows how a failure that occurred can be detected using only counters for different benchmarks - (a) Basicmath, (b) Bitcount, (c) QSort and (d) Dijkstra

These findings provide justification for our hypothesis that tracing HPCs in an interval can be used for anomaly detection in embedded systems.

V. Conclusion

This paper has investigated the use of *Hardware Performance Counters* (HPCs) as a means to extract the features of a system for system reliability monitoring. From the results, we conclude that tracing the HPCs data in a time interval is much more beneficial than looking at final data and that the execution profile developed from the data can be used to monitor if there is any error in the system. This paper presents the first work in using counters for system reliability monitoring. In summary:

- We have explored the various counters available and decided that the number of committed instructions; the number of function calls; the number of integer instructions; and the number of load instructions can be used to identify the occurrence of errors in a system;
- We developed a methodology to observe the anomalous behaviour using HPCs by creating a fault-free pattern and observing changes in the pattern; and
- Our results show that HPCs can be used to identify anomalous behaviour in a system that leads to failure.

Our research will be further extended to capture the correlation between types of anomalous behaviour experienced by the operating system and the errors that causes those anomalous behaviours. We will develop a statistical model based on the 'expected' execution profile (fault-free model) and from this model, we will further characterise and develop a model for the anomalous behaviour identified in the system. We will look into methods to develop online monitoring of a system where the system will be able to monitor itself for any reliability issues by detecting if there is any deviation from the 'expected' execution profile.

Acknowledgment

This work was supported by Microsoft Azure Research Award number CRM: 0518905.

References

[1] N. Wehn, "Reliability: A cross-disciplinary and cross-layer approach," *Asian Test Symposium*, pp. 496–497, 2011.

[2] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[3] A. DeHon, N. Carter, and H. Quinn, "Final report of ccc cross-layer reliability visioning study," Computing Community Consortium (CCC) Visioning Study, United States, Full Report of Computing Community Consortium (CCC) Visioning Study, 2011. [Online]. Available: http://www.relxlayer.org/FinalReport?action=AttachFile&do=view&target=final_report.pdf

[4] H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," *Computing*, vol. 91, no. 1, pp. 43–73, Jnauary 2011. [Online]. Available: http://dx.doi.org/10.1007/s00607-010-0107-y

[5] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM Computer Survey*, vol. 41, no. 3, pp. 15: 1 –15: 58, July 2009.

[6] A. DeOrio, Q. Li, M. Burgess, and V. Bertacco, "Machine learning-based anomaly detection for post-silicon bug diagnosis," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 491–496.

[7] A. Rahimi, L. Benini, and R. K. Gupta, "Variability mitigation in nanometer cmos integrated systems: A survey of techniques from circuits to software," *Proceedings of the IEEE*, vol. 104, no. 7, pp. 1410 – 1448, July 2016.

[8] S. K. S. Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mswat: Low-cost hardware fault detection and diagnosis for multicore systems," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 122 – 132.

[9] N. K. A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Transactions on Computers*, vol. 54, no. 4, pp. 461 – 475, April 2005.

[10] D. Gizopoulos, M. Psarakis, S. V. Adve, P. Ramachandran, S. K. S. Hari, D. Sorin, A. Meixner, A. Biswas, and X. Vera, "Architectures for online error detection and recovery in multicore processors," in *2011 Design, Automation and Test in Europe (DATE)*. IEEE, 2011, pp. 1–6.

[11] M. Kaliorakis, M. Psarakis, N. Foutris, and D. Gizopoulous, "Accelerated online error detection in many-core microprocessor architectures," in *2014 IEEE 32nd VLSI Test Symposium (VTS)*, April 2014, pp. 1 – 6.

[12] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplication instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63 – 75, Mar 2002.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, March 2005, pp. 243 – 254.

[14] T. A. Alves, S. Kundu, L. A. J. Marzulo, and F. M. G. Franca, "Online error detection and recovery in dataflow execution," in *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, July 2014, pp. 9–104.

[15] C. Malone, M. Zahran, and R. Karri, "Are hardware performance counters a cost effective way for integrity checking of programs," in *Proceedings of the sixth ACM workshop on Scalable trusted computing*. ACM, 2011, pp. 71–76.

[16] W. Mathur and J. Cook, "Toward accurate performance evaluation using hardware counters," in *ITEA Modeling and Simulation Workshop*, Dec 2003.

[17] X. Wang, C. Konstantinou, M. Maniatakos, R. Karri, S. Lee, P. Robison, P. Stergiou, and S. Kim, "Malicious firmware detection with hardware performance counters," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 3, pp. 160–173, July 2016.

[18] V. Jyothi, X. Wang, S. K. Addepalli, and R. Karri, "Brain: Behavior based adaptive intrusion detection in networks: Using hardware performance counters to detect ddos attacks," in *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*, Jan 2016, pp. 587–588.

[19] S. Wang, W. Zhang, T. Wang, C. Ye, and T. Huang, "Vmon: Monitoring and quantifying virtual machine interference via hardware performance counter," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, vol. 2, July 2015, pp. 399–408.

[20] S. Rasoolzadeh, M. Saedpanah, and M. R. Hashemi, "Estimating application workload using hardware performance counters in real-time video encoding," in *Telecommunications (IST), 2014 7th International Symposium on*, Sept 2014, pp. 307–311.

[21] Microsoft, "Get started with azure," World Wide Web, retrieved 2017-02-15. [Online]. Available: https://azure.microsoft.com/en-gb/get-started/

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A.Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1 – 7, Aug 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[23] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 622–629.

[24] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *2001 IEEE International Workshop, Proceedings of the Workload Characterization, 2001. WWC-4*, ser. WWC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14. [Online]. Available: http://dx.doi.org/10.1109/WWC.2001.15