Extensive Evaluation of Programming Models and ISAs Impact on Multicore Soft Error Reliability

Anonymous Author(s)

ABSTRACT

To take advantage of the performance enhancements provided by multicore processors, new instruction set architectures (ISAs) and parallel programming libraries have been investigated across multiple industrial segments. It is investigated the impact of parallelization libraries and distinct ISAs on the soft error reliability of two multicore ARM processor models (i.e., Cortex-A9 and Cortex-A72), running Linux Kernel and benchmarks with up to 87 billion instructions. An extensive soft error evaluation with more than 1.2 million simulation hours, considering ARMv7 and ARMv8 ISAs and the NAS Parallel Benchmark (NPB) suite is presented.

1 INTRODUCTION

Multicore processors are a de-facto components in many industrial segments, including automotive, medical, consumer electronics, and high-performance computing (HPC). The ever-increasing demand for computing capacity and energy efficiency leads to the integration of more and more cores, as evidenced in the middle graph of Figure 1, which shows the growing number of cores in commercial processors over the last decades. As more cores are integrated into the same processor, ensuring a reliable operation of such processing elements as well as taking the inherent performance capability, have become challenging issues.

The occurrence of soft errors in multicore processors manufactured on industry-leading process technology nodes is a significant and growing reliability issue in several domains. As illustrated at the bottom graph of Figure 1, commercial processors based on (10 nm) process node are likely to be available in the market at the coming year. Due to the continuously rising number of transistors (top graph of Figure 1), not only more cores but a vast number of memory cells (e.g., registers, internal memory) have been integrated on a single processor die. The exponential growth of internal elements (e.g., cores, memory cells), coupled with the high clock frequency operation of multicore processors is making them more vulnerable to radiation-induced soft errors [14, 20]. While electronic systems working at ground level are expected to experience at least one soft error per day [10], identifying the most unreliable system functionalities is becoming even more difficult due to the ever-increasing complexity of both software and hardware architectures.

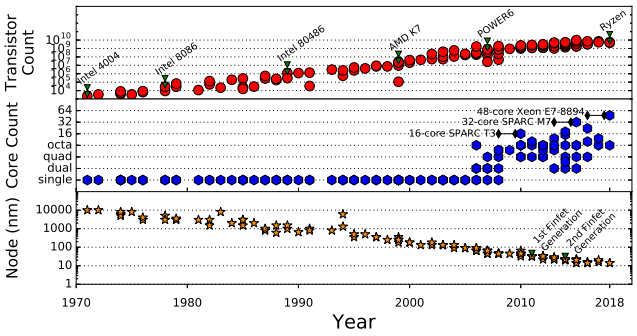


Figure 1: Evolution of commercial processors during the last decades considering the number of transistors (top), number of cores (middle), and associeted technology node (botton) from 1970 to 2018. Information gathered from multiple sources including the ITRS (https://www.itrsgroup.com/).

With 48-core processors available in the market, more efficiently written parallel programs are required to extract the multiprocessing capabilities offered by such systems. Different standard parallelization libraries are available to simplify the process of splitting up the application processing into multiple threads, including Pthreads, OpenMP (OMP), MPI, and Grand Central Dispatch (GCD) [19]. Although applications only benefit from multicore processors performance to a certain point, industrial leaders are investing heavily in the exploration of more efficient programming models and more powerful ISAs, aiming to meet the increasingly large memory and performance requirements of future applications [8, 9]. This problem grows exponentially when multiple of such processors are combined into the same system as both intra-core localities, and intra-processor synchronization might be considered. Today's large-scale HPC systems already exceed 10 million cores [21], and the next generation of computing systems is widely expected to integrate not only more powerful processor cores, which might be homogeneous or heterogeneous, but also much more complex memory organizations.

In this direction, the *contribution* of this paper is to investigate the impact of state-of-the-art ISAs and standard parallelization libraries (e.g., MPI, OpenMP) on the soft error reliability using commercial multicore processors. To achieve an efficient and relevant evaluation, the gem5 is extended to enable the injection of bit-flips on general microarchitectural CPU components (e.g., general-purpose registers). Conducted soft error analysis includes 1,040 million fault injection campaigns, considering multiple versions of the Linux Kernel and real benchmarks employed in both embedded and HPC domains. Further, we propose a data mining tool to correlate the soft error analysis campaigns with profiling information such as gem5 microarchitectural statistics. By finding relationships between fault

DAC'18, June 2018, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

injection campaigns and the application characteristics we intent to reduce the development time of improving the system reliability.

The rest of this paper is organized as follows. First, Section 2 presents related works in virtual platform fault injection simulators. Section 3 details the experimental setup used in this study including the fault injection framework and target software stack. Finally, the results are discussed in Section 4 and final conclusions are presented in Section 5.

2 RELATED WORK

The assessment of multicore systems soft error resilience requires powerful modeling and simulation mechanisms to manage aspects such as resource sharing, memory allocation, and data dependencies. Due to the increasing complexity of such systems, fault injection simulators developed on the basis of virtual platform (VP) frameworks have been considered as an efficient means to assess soft error resilience at an acceptable time. Such frameworks provide high simulation performance, design flexibility (e.g., processor, memory, and peripheral models), and several debugging capabilities. Most fault injection simulators that rely on VP support the injection of bit-flips in memory and register-file components [6, 7, 11, 16, 17, 22]. While the work in [17] supports the fault injection into the processor address generation unit, the authors in [7, 11, 22] explored the impact of injecting bit-flips into the load/store queue. The majority of these works consider either simple (i.e., in-house and bare metal applications) or small scenarios, where only a single-core processor or specific ISA is considered.

To our knowledge, the only work that addresses the impact of parallelization libraries on the soft error reliability is [15]. This work uses a fault injection framework developed on the basis of an instruction-accurate VP simulator [3] and it employs an ARM Cortex-A9 model, which executes three in-house applications on the top of a Linux Kernel. Such adopted applications (e.g., bit count, vector sum, and a 300×300 matrix multiplication) are very small programs, increasing the parallelization libraries API functions exposure to the injected faults, which might overestimate their real impact on the soft error analysis.

Our contribution distinguishes from the previous works in four main aspects:

- First, the impact of two ISAs on multicore soft error reliability is evaluated, considering serial and parallel executions of complex workloads. This is completely ignored in previous works.
- Second, this work uses 130 fault injection scenarios considering real high-performance workloads provided by the NASA NAS Parallel Benchmark suite [2], which includes benchmarks with up to 87 billion instructions.
- Third, we propose a cross-layer reliability evaluation tool which combines profiling information with fault injection results in a single data mining engine.
- Fourth, an extensive multicore soft error evaluation by using several and large scale benchmarks, deploying a *full-system cycle-accurate simulator*.

3 EXPERIMENTAL SETUP

This section details the configuration of the chosen simulator, the fault injector, model, and classification as well as the workflow, software stack, and support tools used to perform this research.

3.1 Simulator

This work adopts the state-of-the-art gem5 simulator [4] for three main reasons: (1) the gem5 source code is open and several extensions have been proposed in the past [1, 18], (2) the gem5 enables *microarchitectural cycle-accurate simulation* in an acceptable time (i.e., 0.4–2 MIPS depending on the application workload), and (3) it supports the current ARM Cortex-A architectures. Moreover, we use the processor model for the ARM Cortex-A9 (ARMv7) and Cortex-A72 (ARMv8) with single, dual, and quad-core variants. Finally, each of the six processor models has a two-level cache memory configured as follows: L1 Instruction 32kB 4-Way Associative, L1 Data 32kB 4-Way Associative, and L2 512kB 8-Way Associative. The gem5 simulator employs Python scripts to control the simulation flow while C++ modules model the microarchitectural components.

3.2 Fault Injector

3.2.1 Fault Model. Developed fault injection framework emulates the occurrence of single-bit-upsets (SBUs) [13] by enabling the injection of bit-flips in a single register or memory address during the execution of a given soft stack. The default fault injection configuration (e.g. bit location, injection time) relies on a random uniform function, which is a well-accepted fault injection technique since it covers the majority of possible faults on a system at a low computation cost [**?**]. Fault injections occur during the target application lifespan (i.e., the OS startup is not subject to faults), which includes OS system calls and parallelization API subroutines arising during this period. This approach allows to identify unexpected application execution errors (e.g., segmentation fault), which are associated to adopted OS components or API libraries. This approach evaluates the application behavior while considering the execution environment, thus exposing unforeseen consequences when compared to standalone implementations.

3.2.2 Fault Classification. We adopted Cho *et al.* [5] classification, which categorizes fault injection outcomes into five groups: **Vanished**, no fault traces are left; **Application Output Not Affected** (ONA), the resulting memory is not modified, however, one or more remaining bits of the architectural state is incorrect; **Application Output Mismatch** (OMM), the application terminates without any error indication, and the resulting memory is affected; **Unexpected Termination** (UT), the application terminates abnormally with an error indication; **Hang**, the application does not finish requiring a preemptive removal.

3.2.3 Workflow. The fault injection campaign follows a four-stage flow. In the *first phase*, Golden Execution, the target architecture is simulated in the absence of faults to extract the reference system behavior, while *phase two* creates a fault target list. In the *third phase*, each gem5 instance simulates the target application, and the included module to perform the fault injection. For each fault injection, the results are compared against the Golden Execution reference, aiming to identify any misbehavior regarding the

number of executed instructions, registers context, and memory state. Lastly, *phase four* assembles all individual reports to create a single database.

3.2.4 Distributed and Parallel Simulation. The fault injection campaigns were performed using an HPC system environment counting with more than 5,000 cores. For this purpose, *phases one* and *two* of the workflow are executed in a local computer, as they are common to all fault injections. An additional step is required to create a series of jobs (i.e., workloads submitted to the HPC) each one comprising a number of fault injections to be computed in the HPC nodes. As such, multiple fault injections can run in parallel (*phase three*). *Phase four* remains unaltered and is run locally after all jobs end. Matching several simulations into a single job improves the HPC scheduling algorithm performance by reducing job management and synchronization overheads.

3.3 Software Stack

3.3.1 Environment. The software stack includes a Linux OS (i.e., the kernel version 3.13 for the ARMv7 and 4.3 for the ARMv8), identical application source code, compilation flags, and cross-compiler (GCC 6.2). Furthermore, we entrust the compiler engine to select the best code optimization according to the target processor by using the flags *-O3* and *-mcpu* set up to either Cortex-A72 or Cortex-A9.

3.3.2 Benchmark. This analysis uses the NAS Parallel Benchmark suite [2] with 29 applications (i.e., 10 Serial, 10 OpenMP, and 9 MPI benchmarks) which was developed by the NASA Advanced Supercomputing Division as a set of programs designed to evaluate the performance of parallel supercomputers. The benchmark was compiled for all six processor models aforementioned in Section 3.1. Some applications do not have an MPI or OpenMP version (e.g., BT and ST applications do not have MPI dual-core implementation), with this, a total of 130 scenarios are available for simulation.

3.4 Data Mining Tool

We propose a cross-layer investigation tool to perform a multi-variable and statistical analysis using the gem5 microarchitectural information (e.g., memory usage, application instruction composition) along with other software profiling tools (e.g., line coverage) that are combined with soft error vulnerability evaluation results (i.e., fault injection campaigns). This work aims to reduce the number of complete fault injection campaigns required during early design space explorations by using software symptoms (e.g., execution time, number of branches) correlated with soft error vulnerabilities to improve the target application reliability.

The analysis process comprises three distinct steps: *First*, the tool collects over one million of single fault injection raw outcomes to create statistical figures (e.g., the percentage of Vanish, Hangs) for each one of the 130 explored scenarios. *Second*, different profiling sources are added into the database, including 200,000 microarchitectural parameters (e.g., CPU utilization, memory statics) and additional information collected using a fast software prototyping tool called Open Virtual Platform (OVPsim) [12] to extract function usage, line coverage, and other parameters not available or accessible into the gem5 simulator. Steps *one* and *two* deploy an exploratory data analysis (EDA) approach to prepare the data for modeling, which involves information acquisition from different sources, data sorting and transformation, and initial statistical analysis.

This approach enables a flexible investigation concerning evaluated variables as new information sources, can be easily included, selected, and conformed to different investigation techniques. *Third*, different databases created during step *one* and *two* are mined to uncover variable relationships in such a way that interesting correlations will become more evident. The promoted tool was constructed using Python that offers a flexible development framework. As it is used by gem5, with our fault injection extension, and many data mining applications, integration between them is effortless. Other components can also be added to analyze different parameters depending on the user's necessities.

Table 1: NPB workload summary.

Description		*Smaller*	*Average*	*Larger*
Simulation Time Single Run (sec.)	ARMv8	35	437	2,134
	ARMv7	163	7,929	42,763
Fault Campaign Run (hours)	ARMv8	77	971	4,742
	ARMv7	363	17,620	95,028
Executed Instructions	ARMv8	41.1×10^6	654×10^6	3.08×10^9
	ARMv7	299×10^6	16.5×10^9	87.4×10^9
				Total
Total Fault Campaign (hours)		ARMv8		82,820
		ARMv7		1,152,160

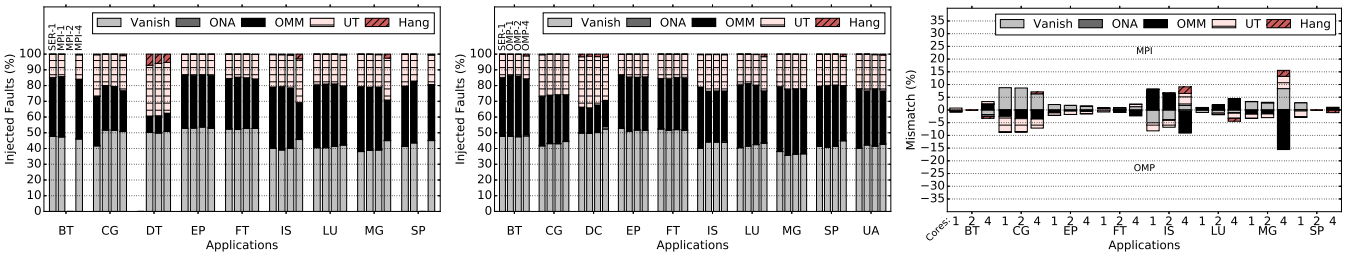
4 RESULTS

We consider the 130 distinct fault injection scenarios where each one suffers 8,000 randomly assigned bit-flips, computing 1,040,000 fault injections which require 1,234,980 simulation hours. Figures 2 and 3 show such scenarios, considering the MPI (Figures 2a and 3a) and the OpenMP (Figures 2b and 3b) applications along with the mismatch of distinct parallelization APIs (Figures 2c and 3c). In this work the mismatch is defined as the sum of absolute differences between each soft error occurrence, such as ONA, OMM.

This investigation explores the application behavior under the presence of faults correlated with distinct profiling parameters, such as execution time, register file size, instruction function calls, composition (e.g., number of branches, loads, and stores), function calls, memory transactions comportment. Further, this analysis aims to expose software symptoms with a direct impact on the application reliability and reveal other parameters, which may not affect soft error vulnerability. By using this indication it is possible to speed-up early designs space explorations.

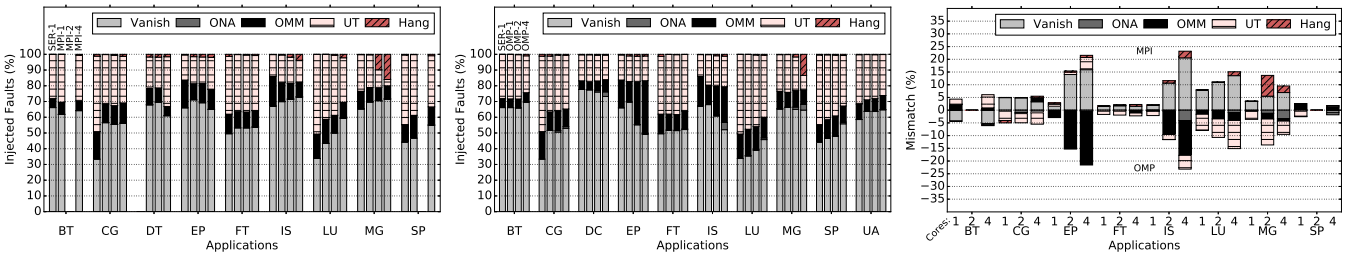
4.1 ISAs Reliability Assessment

4.1.1 Execution Time and Workload. The ARMv7 workload for a single faultless execution has an instruction count that ranges from 250 million to 87 billion, with an average of 16 billion of instructions. In contrast, the 64-bit architecture applications execute



(a) MPI benchmarks. **(b) OMP benchmarks.** **(c) Mismatch.**

Figure 2: NPB fault injections for gem5 using a multicore ARM Cortex-A9 processor (ARMv7).



(a) MPI benchmarks. **(b) OMP benchmarks.** **(c) Mismatch.**

Figure 3: NPB fault injections for gem5 using a multicore ARM Cortex-A72 processor (ARMv8).

in average 654 million instructions, varying from 41 million to 3 billion. Table 1 summarizes the workload regarding simulation time and the number of executed instructions with average, smaller, and larger cases.

Applications executed using the ARMv8 ISA present a significant performance improvement when compared to the ARMv7. In some cases the speedup reaches up to 10 times. This performance gain can be pinpointed to the removal of several legacy features (e.g., fast and multilevel interruptions, conditional instructions) and to significant improvements in the floating-point (FP) unit by adding new specialized instructions and increasing the FP register file. The ARMv7 often resorts to the ARM software FP library to perform some operations and thus increasing execution time. This choice was made automatically by the compiler (Section 3.3.1). The evaluated workload employs HPC scientific applications with some of them heavily depending on FP computation, leading to a significant performance boost. The executed instruction count for each application where the average value reduces from 16 billion (ARMv7) to 654 million (ARMv8) instructions (Table 1). A shorter execution time improves the ARMv8 mean time between failures (MTBF) as it has a smaller probability of being stroke by a radiation event for a given particle fluence[1].

4.1.2 Register File Size. The new 64-bit ISA also enlarges the integer register-file, from 16 to 32 registers, increasing the number of possible targets for fault injection by a factor of four. However, the compiler algorithm uses a reduced fraction of the available registers for load/store and control flow operations leaving other registers for global variables or unused. As in this experiment each register suffers an identical number of fault injections, critical registers (e.g., program counter, stack pointer, those used on load/store and control flow operations) are less likely to face faults in the ARMv8 rather than in the ARMv7.

4.1.3 Branches and Function calls. The *Hang* error occurs when the target application control flow is severely affected by transient faults, in most cases, leaving the algorithm in an infinite loop. Analyzing individual parameters not always expose direct relationships between profiling data and fault injection campaigns. For example, the mean branch composition from the total executed instructions is 19.24% ($\sigma = 0.21$), 14.08% ($\sigma = 0.56$), 17.65% ($\sigma = 0.03$), and 12.01% ($\sigma = 0.36$) considering the four macro scenarios MPI V7, OMP V7, MPI V8, and OMP V8, where σ is the standard deviation. While the ARMv8 displays a 2% decrease in the mean branch occurrence compared with the 32-bit architecture, the application behavior under fault influence does not show any meaningful impact. Additionally, *function calls* variation also does not display any distinctive link with Hangs incidence. By combining both figures, nonetheless, is possible to uncover a correlation between this new index value (i.e., number of function calls times number of branches) with the Hang incidence after comparing the 130 scenarios. Table 2 exemplifies this behavior using the IS application as a case study, note that this new index value and the Hang percentage increases simultaneously, a behavior observable trough several scenarios. The ARM ISA (i.e., ARMv7 and ARMv8) use distinct instructions to compare the conditional statement (e.g., cmp) and another to perform the control flow branching, while function calls use unconditional branches (e.g., jumps) in conjunction of argument registers.

[1]The number of radiant-energy particles incident on the target system surface in a given period of time

Table 2: Hang occurrence compared with the normalized function calls multiplied branches (F*B).

Scenario	Parameter	Number of Cores		
		Single	Dual	Quad
IS MPI V7	Hang (%)	0.413	0.625	3.000
	Branches	56.0×10^6	58.0×10^6	196×10^6
	F. Calls	22.6×10^6	23.1×10^6	26.9×10^6
	Index F*B	1.000	1.024	1.700
IS OMP V7	Hang (%)	0.288	0.313	0.400
	Branches	54.1×10^6	54.3×10^6	54.7×10^6
	F. Calls	21.7×10^6	21.7×10^6	21.7×10^6
	Index F*B	1.000	1.001	1.002
IS MPI V8	Hang (%)	0.438	1.850	3.800
	Branches	11.2×10^6	15.9×10^6	17.6×10^6
	F. Calls	2.85×10^6	3.35×10^6	4.84×10^6
	Index F*B	1.000	1.302	1.799
IS OMP V8	Hang (%)	0.225	0.925	1.175
	Branches	7.99×10^6	9.05×10^6	9.50×10^6
	F. Calls	1.81×10^6	2.05×10^6	2.06×10^6
	Index F*B	1.000	1.172	1.194

Table 3: ARMv7 Memory transactions and soft error classification for selected scenarios.

	Scenario	Vanish +OMM +ONA	UT	Mem. Inst. (%)	RD/WR Ratio
1	MG MPIx1	78	22	15.8	1.18
2	MG MPIx2	78	22	16.3	1.12
3	MG MPIx4	70	30	22.5	2.83
4	IS MPIx1	80	20	18.0	0.85
5	IS MPIx2	80	20	19.0	0.83
6	IS MPIx4	70	31	26.0	2.73

4.1.4 Memory Transactions. UTs (i.e., unexpected terminations) originates from OS *segmentation fault* exceptions which means that the program has attempted to access an area of memory outside its permissions. At instruction level, the address generation of memory access operations (e.g., load and stores) is compromised by transient faults in the source registers leading to wrong address calculations. The reduced number of ARMv7 registers to perform address calculations leads to the use of load/store templates by the compiler to diminish the computational cost of register recycling. In other words, the ARMv7 compiler continuously utilizes the same register to perform memory transactions (e.g., R0–3 and SP). As consequence of this behavior, increasing the number load/store operations can lead to a more significant UT occurrence in the target application using an OS on top of the ARMv7 processor.

Table 4: ARMv8 Memory transactions and soft error classification for selected scenarios.

	Scenario	Vanish +OMM +ONA	UT	Mem. Inst. (%)	RD/WR Ratio
A	LU OMPx1	57	48	29	1.9
B	LU OMPx2	59	45	27	1.9
C	LU OMPx4	67	40	22	1.9
D	SP OMPx1	57	42	35.1	1.5
E	SP OMPx2	59	40	34.0	1.5
F	SP OMPx4	70	32	28.5	1.5
G	FT MPIx1	62	37	25.7	1.00
H	FT MPIx2	62	37	24.6	0.95
I	FT MPIx4	62	36	23.7	0.95

Table 3 shows the soft error results (e.g., Vanish, UT, Hangs) alongside the memory access figures for some examples of the behavior mentioned above. By increasing the percentage of memory transactions (i.e., load and stores instructions) in applications such as MG and IS increases the UT ratio. For example, MG application memory-oriented operations for single and quad-core processors are 15%, and 22% while the UT occurrence increases from 22% to 30%. Further, increasing the core count alone does not reduce the UT percentage as is possible to note by comparing scenarios (1, Table 3) against (2) where both have similar memory instruction occurrence.

The 64-bit architecture exhibits a similar behavior considering FP memory transactions, supporting the claim above that wrong address calculation related to memory access, as FP instructions are exclusively used for computation and not for control flow operations (e.g., branches and jumps). Table 4 displays nine scenarios (A-I) of soft error analysis and FP memory figures. Reducing the memory transactions participation from the total number of executed instructions for LU (A-C) and SP (D-F) applications show a UT occurrence reduction trend. Scenarios (G-I) reinforce this hypothesis by demonstrating that a constant memory-oriented instruction incidence leads to a regular UT percentage.

4.2 Parallelization API

The OpenMP library uses a series of the fork and joins approach to parallelize loop statements (e.g., *for*, *while*) where the API automatically create children threads, being suitable for shared memory. In contrast, the MPI standard is adequate to distribute systems due to the use of a message-oriented parallelization technique which requires the direct user parallelization regarding thread creation and communication. Figures 2c and 3c display the mismatch comparing the MPI and OpenMP applications whenever present for both APIs for the ARMv7 and ARMv8 respectively.

4.2.1 Serial vs APIs. When we compare the serial implementation with either parallelization libraries on both architectures, some patterns can be observed. In ARMv7 MPI, only CG has a small improvement in the number of UTs, while in IS and MG the number

of UTs and Hangs increases. Considering the OpenMP versions, no significant variation can be found. For the 64-bit application set, CG, LU, MG, SP, and UA the number of UTs diminishes. Further, CG application maintains the number of UTs when the number of cores increases. The same cannot be said about the other application, where the number of UTs diminishes with the increase in core count. Other applications have negligible variations.

4.2.2 Vulnerability Window. Within a software stack, some components are more critical than others to the system correct behavior. For example, targeting a thread scheduling function with faults has a potentially more hazardous effect on the system reliability than a purely arithmetic code portion. By comparing these critical functions active periods against application execution time it is possible to define a time interval called *vulnerability window*, which varies with the number of calls and executions of the function. Using the NBP benchmark suite provides a real high-performance workloads, enabling a more accurate evaluation of the OpenMP and MPI libraries impact on the system reliability. Due to its reduced vulnerability window, the parallelization mechanism has a limited effect on the final reliability assessment, less than 23% in the worst case.

From the 44 possible comparisons between the MPI and OpenMP scenarios, in 38 the MPI has a higher masking rate (i.e., executions without any errors) due to two main reasons: *First*, MPI applications have a better workload balance among the used cores, in other words, the number of executed instructions per core is very similar. For instance, the average difference concerning executed instructions per core is around 4% for both ARMv7 and ARMv8 considering MPI applications, while the OpenMP variation reaches up 16%. As the OpenMP does not fully utilize the available cores due to the fork/join parallelization approach where a loop statement executes in parallel and other code portions hastily. By contrast, the MPI has individual and independent working threads for each running core providing a better workload balance during its execution. Whenever a core is sub-utilized, it executes a thread scheduling policy and when no thread is suitable the core waits in a sleep mode. By consequence the kernel probability to suffer a transient fault increases, as the scheduling is more often executed. *Second*, OpenMP benchmarks have a smaller execution time, 16% in average, compared against the MPI applications. By consequence, diminishing the vulnerability window of the MPI inner-functions when comparing against the OpenMP. Further, the application longer execution increases the chance of the injected fault being erased due to software and microarchitectural masking mechanisms.

5 CONCLUSION AND FUTURE WORKS

This work has investigated the impact of the parallelization APIs and the ISA in the overall software stack reliability by comparing serial, OpenMP, and MPI implementations of the same benchmarks using both ARMv7 and ARMv8 architectures. This work amasses hundred of thousands of profiling and microarchitectural parameters alongside more than a million hours of simulation worthy of fault injection. To explore such large trove of collected data we propose a cross-layer investigation tool to mine relationships between soft error vulnerability analysis and other application statistics. MPI applications have a slightly larger masking rate, in other words, suffer less from the fault injection. Fault campaigns show a smaller incidence of the parallelization API in the overall system reliability due to the limited time ratio of those libraries in comparison with the total execution time. MPI is more prone to deadlocks due to failed communication, and OpenMP parallelization paradigm increases the chance of unexpected terminations. When comparing ARMv7 and ARMv8, the former suffers greatly in performance from the use of software FP arithmetics, which was set automatically by the compiler. Future works could explore the relationship of compiler flags and application behavior regarding soft errors. Further, we intend to include more sophisticated data mining algorithm, making possible the correlation of even more data points (e.g., Memory Management Unit statistics, number of power state transitions), and include new profiling inputs.

REFERENCES

[1] M. Alian et. al. 2016. pd-gem5: Simulation Infrastructure for Parallel/Distributed Computer Systems. *IEEE Computer Architecture Letters* 15, 1 (Jan. 2016), 41–44.

[2] D. H. Bailey et. al. 1991. The NAS parallel benchmarks summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. 158–165.

[3] F. Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (2005), 41–41.

[4] N. Binkert et. al. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7.

[5] H. Cho et. al. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *2013 50th ACM / EDAC / IEEE Design Automation Conference (DAC)*. 1–10.

[6] F. de Aguiar Geissler et. al. 2014. Soft error injection methodology based on QEMU software platform. In *Test Workshop - LATW, 2014 15th Latin American.*

[7] M. Didehban et. al. 2016. nZDC: A Compiler Technique for Near Zero Silent Data Corruption. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, 48:1–48:6.

[8] Massive Parallelism for Mission-Critical Applications. 2017. (2017). https://www.intel.com/content/www/us/en/processors/itanium/itanium-9500-massive-parallelism-mission-critical-computing-paper.html

[9] J. Goodacre et. al. 2005. Parallelism and the ARM instruction set architecture. *Computer* 38, 7 (July 2005), 42–50.

[10] T. Granlund et. al. 2003. Soft error rate increase for new generations of SRAMs. *IEEE Transactions on Nuclear Science* 50, 6 (Dec. 2003), 2065–2068.

[11] Q. Guan et. al. 2016. Design, Use and Evaluation of P-FSEFI: A Parallel Soft Error Fault Injection Framework for Emulating Soft Errors in Parallel Applications. In *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques (SIMUTOOLS'16)*. ICST, Brussels, Belgium, 9–17.

[12] Imperas. 2017. Open Virtual Platforms (OVP). (2017). http://www.ovpworld.org/

[13] K. Johansson et. al. 1999. Neutron induced single-word multiple-bit upset in SRAM. *IEEE Transactions on Nuclear Science* 46, 6 (Dec. 1999), 1427–1433.

[14] S. Mukherjee. 2008. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[15] G. Rodrigues et. al. 2017. Analyzing the Impact of Fault Tolerance Methods in ARM Processors under Soft Errors running Linux and Parallelization APIs. *IEEE Transactions on Nuclear Science* PP, 99 (2017), 1–1.

[16] F. Rosa et. al. 2015. A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability. In *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. 211–214.

[17] S. K. Sastry Hari et. al. 2014. GangES: Gang Error Simulation for Hardware Resiliency Evaluation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 61–72.

[18] Y. S. Shao et. al. 2016. Co-designing accelerators and SoC interfaces using gem5-Aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12.

[19] D. Shekhar T.C et. al. 2011. Comparison of Parallel Programming Models for Multicore Architectures. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 1675–1682.

[20] M. Snir et. al. 2014. Addressing Failures in Exascale Computing. *Int. J. High Perform. Comput. Appl.* 28, 2 (May 2014), 129–173.

[21] TOP500 Supercomputer. 2017. (2017). https://www.top500.org/

[22] K. Tanikella et. al. 2016. gemV: A validated toolset for the early exploration of system reliability. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 159–163.