DOI: 10.1016/j.microrel.2019.06.038

Analysis of neutron sensitivity and data-flow error detection in ARM microprocessors using NEON SIMD extensions

A. Lindoso, M. Garcia-Valderas, L. Entrena

Abstract—**This work analyzes the sensitivity to neutrons of SIMD (Single Instruction Multiple Data) microprocessor extensions. To this purpose, the ARM SIMD coprocessor (NEON™) was selected as a case study and neutron radiation experiments were performed on a commercial device running NEON software. In addition, we analyze the benefits of using the NEON coprocessor as a means to efficiently implement data-flow hardening approaches. Experimental results show that SIMD extensions have a great potential to improve performance and reduce the overheads associated to software data-flow hardening.**

Index Terms— **Neutrons, SIMD, ARM, NEON, fault tolerance, software hardening.**

I. Introduction

MICROPROCESSORS currently require high-end capabilities in order to execute complicated and tightly-constrained software applications. Algorithms are increasing in complexity and performance needs in all application fields, even in those that must satisfy high reliability requirements. To cope with the increasing demand for computational power, microprocessors gradually include more sophisticated architectures and new features. Among these features, SIMD (Single Instruction Multiple Data) coprocessors are a noteworthy extension that can be found in advanced microprocessors. Examples of these coprocessors are NEON™ in the case of ARM microprocessors [1] or SSE [2] in the case of Intel and AMD microprocessors.

An SIMD coprocessor is intended to improve performance for algorithms that perform the same computations over large data sets. Examples of this type of algorithms can be found in digital signal processing or image processing. For instance, image filtering usually involves multiply-accumulate operations that must be repeated for every single image pixel. To this purpose, an SIMD coprocessor contains a parallel Arithmetic-Logic Unit (ALU) that is able to process multiple data with a single instruction. Data are stored in a special register file with wider registers, where each register can allocate several data, and a special set of SIMD instructions is used to implement operations on these registers. Notably, SIMD instructions are similar to regular instructions, so that they do not add any particular fetching or decoding overhead. Thus, using a SIMD coprocessor, computations over large data sets can be significantly accelerated through data parallelism.

In this context, the objective of this work is twofold. On the one hand, we want to evaluate the susceptibility of the ARM SIMD coprocessor (NEON) under neutron irradiation. The ARM architecture is currently one the favourite choices in the embedded market and it is also being used in high-reliability sectors, such as automotive [3-4] or medical [5-6]. In previous work, we have evaluated the sensitivity of NEON to low-energy protons [7]. In this work we analyse the behaviour of NEON under neutrons, which have more interest for ground applications. Radiation experiments were performed at Los Alamos National Laboratory's (LANL) Los Alamos Neutron Science Center (LANSCE).

On the other hand, we want to study the benefits of using the NEON coprocessor as a means to implement data-flow hardening techniques. Data-flow hardening techniques typically rely on the replication of variables in the sofware [8]. Operations are repeated for each data replica and then the results are compared to detect or correct errors. Repeating computations entails large overheads in performance and memory usage. The capability of a SIMD coprocessor to concurrently execute instructions on multiple data can be exploited to reduce these overheads. To this purpose, data can be replicated in the NEON registers, so that the processing of the replicas can be executed in parallel.

This paper is organized as follows. Section II introduces the SIMD extensions. Section III presents a novel SIMD application to data-flow error detection. Section IV presents the experimental results and finally Section V summarizes the conclusion of this work.

II. SIMD extensions

SIMD extensions provide an efficient way of performing the same operation over a data set. The conventional serial microprocessor architectures are not efficient for this type of computations. Assuming the data needed by a computation is in the register file, each instruction must be fetched, decoded, executed and finally store the result in a register or in memory. If the next instruction is computing the very same operation but with different data, the process must be repeated. Instead, in a

SIMD processor a single instruction can perform an operation over several data in parallel. Results are also stored in parallel register files, so that following SIMD instructions can work seamlessly on the parallel data set.

Many advanced microprocessors include SIMD extensions in their architectures, which are typically implemented as coprocessors. These coprocessors are able to perform the very same operation over a data set of configurable size with a single instruction. They have a specific register file that contains wider registers. Taking as an example the ARM architecture (32 bits), its SIMD coprocessor (NEON) includes 128-bit registers [9]. These registers can store a vector of data and support several data types. When the data type is small, that NEON vector contains a larger amount of elements. For instance when 32 bit data is used, the 128 bit NEON vector can allocate 4 elements. If data size is reduced to 8 bits, the number of elements per vector is increased to 16. The elements of SIMD vectors are named lanes.

When an operation is performed over a NEON vector, this operation usually involves the utilization of all the elements or lanes of the vector. For instance, addition of NEON vectors is performed with the VADD assembly instruction. Following is a complete example of the use of this instruction:

```
VADD.I32  D0, D0, D1
```

When this instruction is executed, all the lanes of NEON vector D0 are added with the corresponding lanes of NEON vector D1 and stored back in vector D0. Carry is not considered between lanes. In this example, the selected data type is 32 bit integer (I32), meaning that with one NEON instruction four additions of 32 bit integers are performed.

SIMD coprocessors require loading data into their register file in an efficient manner. To this purpose, specific SIMD instructions and data type qualifiers are provided to support load, store and move operations with SIMD coprocessor registers. Depending on the data structure and the used algorithm, the programmer must devise an optimal strategy to arrange data in the most efficient way for the processing.

NEON programming can be done in three different ways: using automatic compilation options, using specific C functions, called *intrinsics*, or directly coding in assembly using NEON assembly instructions. To use automatic compilation, the code must be adapted so that the compiler is able to detect code structures that are suitable for implementation with NEON instructions. The generated code must be checked to verify if the compiler has been effective in the NEON usage.

Assembly coding leads to the best results in terms of performance, but it also involves the highest development effort. Intrinsics are a good trade-off between performance improvement and design effort. Intrinsics make possible to use the NEON instruction set in a very easy way, because they free the programmer from low-level decisions such as register mapping while detailed NEON operations can be specified. With intrinsics, programmers can have low-level control of the entire NEON processing including the load of NEON vectors, the arrangement of NEON operations and the storing of NEON results back.

III. Data-flow error detection using SIMD

Data-flow software hardening techniques usually involve a significant performance decrease [10]. These techniques are generally based on duplicating all data and operations. Then, for every computation, the affected variables and their copies are compared. If a discrepancy is detected, a data error is triggered [8]. This technique has two clear disadvantages: it increases the memory occupation, due to duplicated data, and decreases the performance. The decrease in performance is due to the duplicated data flow, which requires duplicating the processing and comparing the results. To reduce the performance penalty, some authors propose using a tunable number of comparisons or a configurable reliability level by selecting a subset of variables to be protected by duplication [11].

SIMD is intended to accelerate repeated computations and this capability can be exploited for duplicated data. Data duplication is based on storing each variable twice and performing the same operations with each copy of the variables. As the operations performed on the duplicated variables are the same, they are perfectly suited to be implemented with SIMD instructions. To this purpose, duplicated variables are stored in the SIMD register file and then processed with SIMD instructions. If the SIMD coprocessor is used efficiently, the decrease in performance for the data-flow duplication can be overcome.

The technique we are proposing takes advantage of a part of the microprocessor that is not always in use. Many applications are not suitable for parallelization and cannot take advantage of the SIMD coprocessor. In this case, the proposed technique can be implemented with minimal overhead. In the case of software that uses the SIMD coprocessor, only a subset of functions are usually affected. The proposed technique can still be used at the expense of proportionally reducing the speed-up achieved by the SIMD accelerated functions.

Loading data into the SIMD register file and retrieving results from it are performance critical processes that must be carefully implemented. There are several choices to manage the loading of NEON vectors that can ease the process. To maximize the SIMD parallel processing capacity, NEON vectors must use all lanes and load/store must be as less frequent as possible. That typically requires rearranging original data and copies in a smart way. The most efficient load for a NEON vector consist in directly accessing an array and store array elements in different lanes of the NEON vector. In our implementations we have merged the original data and the duplicated data in one single array to enable an optimal load of NEON vectors.

Note that the proposed technique cannot accelerate the comparison of original data and duplicated data. SIMD instructions are intended for parallel processing without conditional evaluation that could decrease the performance speedup.

IV. Experimental results

To evaluate the neutron sensitivity of NEON-based applications, a neutron radiation campaign was performed in September of 2017 at Los Alamos Neutron Science Center (LANSCE), Los Alamos National Laboratory (LANL). LANSCE facility has a white neutron source which emulates the energy spectrum of the atmospheric neutron flux. In the following subsections we describe the experimental setup and the results obtained in this campaign.

A. Experimental setup

For the experiments we have used eight Zybo boards [12], each one containing a XC7Z010 Zynq All Programmable System-on-Chip (APSoC) device from Xilinx [13]. Xilinx Zynq technology integrates hard-core ARM processors within an SRAM-based FPGA. The selected device contains a dual core ARM CORTEX™-A9 with SIMD capabilities, i.e. one NEON™ coprocessor per core. For the experiments, only one ARM core was used at the nominal 650 MHz clock frequency. The programmable logic was not used in this work.

In order to maximize the number of events, all the eight boards were exposed to the beam during the complete duration of the radiation campaign. The campaign took six days.

Two external hosts located outside the beam were used to collect and classify the observed errors and to restart the Devices Under Test (DUTs) when needed. Each external host controlled four Zybo boards, each through a separated USB connection. To restart a device when an error was observed, the external hosts control a set of relays that switch off the power and then switch it on again. The external monitors also include watchdog timers that trigger the restart of a DUT when it does not respond for some predefined amount of time, which indicates that the processor is lost.

The experiments were performed with several versions of a matrix multiplication benchmark: conventional matrix multiplication code (Mmult), matrix multiplication using NEON coprocessor (Mmult_N) and matrix multiplication with hardened data-flow using NEON coprocessor (Mmult_DN). The data-flow hardening was implemented using the technique described in section III in which the NEON coprocessor processes duplicated data with NEON instructions. All benchmarks used 20x20 data arrays of 32 bit input data.

Table I shows the number of cycles and execution time for all the benchmarks. We have also included in the table the case of the hardened data-flow matrix multiplication benchmark (Mmult_D) implemented with conventional instructions instead of the NEON coprocessor. This benchmark was not irradiated and it is only included here for the sake of performance comparison.

The data in Table I shows that the use of NEON improves the performance by 2.4 times with respect to the conventional implementation. If the data-flow is duplicated using NEON (Mmult_DN), the performance decreases, but it is still 1.67 times better than the basic Mmult implementation. On the contrary, the duplication of the data-flow in a conventional manner (Mmult_D) introduces a high performance penalty (3.63 times w.r.t. Mmult). This is due to the duplication and comparison of the computations and to the need to reduce compilation optimization in order not to destroy the duplicated data-flow software structure.

TABLE I: PERFORMANCE

Benchmark	#cycles	Execution time (µs)
Mmult	50,442	77.60
Mmult_N	20,877	32.12
Mmult_DN	29,979	46.12
Mmult_D	183,190	281.83

A. Experimental Results

Tables II, III and IV summarize the results of the experiments. They show the observed errors for each of the tested benchmarks, Mmult, Mmult_N, and Mmult_DN, respectively. Error categories reported in the tables are the following:

- Hang errors: The microprocessor cannot continue with normal execution of the benchmark any longer and needs external intervention. An error has provoked that the microprocessor is stuck at an infinite loop or has produced an abnormal termination. Hang errors are detected by the watchdog in the external host. After collecting and storing the error, the external host restarts the DUT.
- Silent Data Corruption (SDC) errors: After every iteration of the executed algorithm, the result is compared with a golden result. SDC errors are detected when there is any discrepancy between the actual and the golden result. SDC errors are detected by the software and reported to the external host for recording and classification.
- Communication error (Comm.): The microprocessor cannot perform communication in a proper way with the external host. Although this is not an indication of a microprocessor error, the DUT must be restarted to be able to report its state. Tables II, III and IV show that this type of error represents a negligible fraction of the observed errors.
- Detected Data error (Det.): This category detects data errors and it is only available for the NMmult_DN benchmark (Table IV), which is the only benchmark that implements data-flow hardening.

The comparison of the results in Tables II and III show that the Mmult conventional version (Mmult) and the NEON Mmult version (Mmult_N) present approximatively the same error distribution (see column 3, errors/total errors).

Table IV shows the error distribution for the NEON-based data-flow hardening technique. Simply using duplicated data in the NEON coprocessor, 30% of the observed errors were detected. This is a reasonable result taking into account that only the data-flow was hardened. For a complete solution, the proposed techniques must be combined with other hardening techniques [15], particularly for the control-flow, which usually accounts for the majority of errors in microprocessors [14].

There are solutions for control-flow error detection which can be implemented with low performance penalty [15]. However, data-flow hardening is usually implemented in software at the expense of severely reducing performance. The proposed NEON-based approach can significantly alleviate this performance penalty.

Experimental results also show that Hang errors are much less frequent than SDC errors for all Mmult benchmarks. This result corresponds to the characteristics of the matrix multiplication algorithm, because it is intensive in data. Thus, it can be expected that most errors show up as SDC errors.

TABLE II: MMULT experimental results

Error type	Observed errors	Errors/total errors
Hang	20	10.64%
SDC	168	89.36%
Comm.	0	0.00%
Total	188	100%

TABLE III: MMULT_N experimental results

Error type	Observed errors	Errors/total errors
Hang	39	16.32%
SDC	192	80.33%
Comm.	8	3.35%
Total	239	100%

TABLE IV: MMULT_DN experimental results

Error type	Observed errors	Errors/total errors
Hang	17	6.16%
SDC	172	62.32%
Comm.	4	1.45%
Det.	83	30.07%
Total	276	100%

Table V reports the fluence and the cross section for all benchmarks. Mmult_N increases the cross section w.r.t. Mmult by a factor of 1.6. However, this increase is offset by the performance improvement (2.4 times). This result is in agreement with previously reported results using low-energy protons [7]. The smaller cross-section of Mmult is because it uses a small amount of registers. If the ARM register file is fully used, the cross section is no better than that of Mmult_N (the details of this discussion will be provided in the final paper). The error rate increases for the Mmult_DN benchmark because it contains duplicated variables and uses more resources. Thus, the cross section is higher ($5.02\cdot10^{-9}$ cm^2) taking into account all errors. However, it reduces to $3.84\cdot10^{-9}$ cm^2 if only the undetected errors are considered.

V. Conclusions and Future Work

This work explores the reliability of SIMD extensions under neutron irradiation. In particular we have used the NEON coprocessor, which is the SIMD extension of the ARM architecture. We have also proposed a new approach to implement data-flow hardening that makes an effective use of the NEON coprocessor.

Experimental results show that the use of NEON can provide a good balance between performance and cross-section. In addition, it can be used to implement data-flow hardening techniques in an effective way to significantly reduce the performance penalty caused by these techniques. Future work is intended to combine the proposed approach with other hardening techniques towards the development of complete microprocessor hardening solutions.

Additional experimental results with more benchmarks will be provided in the final paper.

TABLE IV: CROSS SECTION

Benchmark	Fluence (n/s·cm^2)	Cross section (cm^2)
Mmult	$7.15\cdot10^{10}$	$2.63\cdot10^{-9}$
Mmult_N	$5.58\cdot10^{10}$	$4.28\cdot10^{-9}$
Mmult_DN	$5.02\cdot10^{10}$	$5.50\cdot10^{-9}$ / $3.84\cdot10^{-9}$

References

[1] ARM Inc, “ARM architecture reference manual”, 2000.
[2] Intel Inc., “Intel® 64 and IA-32 Architectures, Software Developer’s Manual, Volume 1: Basic Architecture”, 2016.
[3] X. Iturbe, B. Venu; J. Jagst; E. Ozer; P. Harrod; C. Turner; J. Penton, “Addressing Functional Safety Challenges in Autonomous Vehicles with the Arm Triple Core Lock-Step (TCLS) Architecture”, IEEE Design & Test, : DOI 10.1109/MDAT.2018.2799799.
[4] I. Stoychev, P. Wehner, J. Rettkowski, T. Kalb, D. Göhringer, J. Oehm, “Sensor data fusion with MPSoCSim in the context of electric vehicle charging stations”, IEEE Nordic Circuits and Systems Conference (NORCAS), pp 1-6, 2016.
[5] C. Wang, J. Zhou, L. Liao, J. Lan, J. Luo, X. Liu, M. Je, “Near-Threshold Energy- and Area-Efficient Reconfigurable DWPT/DWT Processor for Healthcare-Monitoring Applications”, IEEE Transactions on Circuits and Systems II: Express Briefs, Vol: 62, Issue: 1, pp: 70 – 74, 2015.
[6] Y. Fu; F. Zhang; X. Ma; Q. Meng, “Development of a CPM Machine for Injured Fingers”, 2005 IEEE Engineering in Medicine and Biology 27th Annual Conference, pp: 5017 – 5020, 2005.
[7] A. Lindoso; M. García-Valderas; L. Entrena; Y. Morilla; P. Martín-Holgado, “Evaluation of the suitability of NEON SIMD microprocessor extensions under proton irradiation”, IEEE Transactions on Nuclear Science, DOI:10.1109/TNS.2018.2823540, 2018.
[8] M. Rebaudengo, M. S. Reorda, M. Torchiano, M. Violante, “Soft-error detection through software fault-tolerance techniques,” Proc. Intl. Symp. on Defect and Fault Tolerance in VLSI Systems, pp. 210–218, 1999.
[9] ARM Inc., “NEON programmer’s guide”, 2013.
[10] M. Nicolaidis, “Soft errors in modern electronic systems”, Springer New York, 2011.
[11] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, “A C/C++ source-to-source compiler for dependable applications”, Proceeding International Conference on Dependable Systems and Networks. DSN 2000, pp: 71 – 78, 2000.
[12] Digilent Inc, “Zybo reference manual”, 2014.
[13] Xilinx Inc.: “Zynq-7000 All Programmable SoC: Technical Reference Manual”, UG585, 2016.
[14] L. Parra, A. Lindoso, M. Portela-Garcia, L. Entrena, B. Du, M. Sonza Reorda, L. Sterpone, “A New Hybrid Nonintrusive Error-Detection Technique Using Dual Control-Flow Monitoring”, IEEE Transactions on Nuclear Science, vol. 61, no. 6, pp. 3236-3243, Dec. 2014.
[15] A. Lindoso, L. Entrena, M. García-Valderas, L. Parra. "A hybrid fault-tolerant LEON3 soft core processor implemented in low-end SRAM FPGA". IEEE Transactions on Nuclear Science, vol. 64, no. 1, pp. 374-381, Jan. 2017.