rACE: Reverse-Order Processor Reliability Analysis

Athanasios Chatzidimitriou Dimitris Gizopoulos
Department of Infromatics and Telecommunications
University of Athens
Athens, Greece
{achatz | dgizop}@di.uoa.gr

***Abstract*—Modern microprocessors suffer from increased error rates that come along with fabrication technology scaling. Processor designs continuously become more prone to hardware faults that lead to execution errors and system failures, which raise the requirement of protection mechanisms. However, error mitigation strategies have to be applied diligently, as they impose significant power, area, and performance overheads. Early and accurate reliability estimation of a microprocessor design is essential in order to determine the most vulnerable hardware structures and the most efficient protection schemes. One of the most commonly used techniques for reliability estimation is Architecturally Correct Execution (ACE) analysis.**

ACE analysis can be applied at different abstraction models, including microarchitecture and RTL and often requires a single or few simulations to report the Architectural Vulnerability Factor (AVF) of the processor structures. However, ACE analysis overestimates the vulnerability of structures because of its pessimistic, worst-case nature. Moreover, it only delivers coarse-grain vulnerability reports and no details about the expected result of hardware faults (silent data corruptions, crashes). In this paper, we present reverse ACE (rACE), a methodology that (a) improves the accuracy of ACE analysis and (b) delivers fine-grain error outcome reports. Using a reverse-order tracing flow, rACE analysis associates portions of the simulated execution of a program with the actual output and the control flow, delivering finer accuracy and results classification. Our findings show that rACE reports an average 1.45X overestimation, compared to Statistical Fault Injection, for different sizes of the register file of an out-of-order CPU core (executing both ARM and x86 binaries), when a baseline ACE analysis reports 2.3X overestimation and even refined versions of ACE analysis report an average of 1.8X overestimation.

Keywords—reliability, dependability, microarchitecture, simulation, ACE analysis.

I. Introduction

Technology advances have enabled designers to increase the performance and complexity of modern microprocessors. The continuous shrinking of transistors, however, has a negative impact on the device reliability. Denser chips tend to be more vulnerable to cosmic radiation, environmental conditions, voltage fluctuations, aging, etc., which are responsible for the introduction of faults on a system [1] [2]. These faults can harm the correct operation and cause malfunctions, such as output corruptions or system crashes. Architectural Vulnerability Factor (AVF) [3] is a metric that quantifies the vulnerability of a system towards faults and expresses the probability of a fault to affect the correct execution of a program on a system.

This work has been funded by the European Union through the CLERECO project (FP7 Grant 611404), the UniServer project (H2020 Grant 688540), a Tetramax-Bilateral-TTX-2 grant, and by Cisco Research.

New error rates and types come with every manufacturing node and require additional effort in order to make a design comply with reliability constraints. Computer architects apply protection mechanisms that aim to detect and even repair errors at the circuit level. However, these mechanisms come at a cost, in terms of area, power and performance overheads [4]. By adding protection mechanisms, a design improves the level of reliability but is negatively affected in other characteristics. Thus, it is important to integrate a “just-right” protection and not over- or under-protect the system. The only way to achieve this is to estimate early and accurately the vulnerability of the microprocessor.

There are several reliability estimation methodologies aiming to aid reliably-related decisions, by providing early estimates of the vulnerability of a design. All estimation methodologies are performed on top of a design model and the results guide design decisions and revised designs have again to be assessed for vulnerability. This process requires time for every iteration that is relative to how late the design was assessed. Late design changes for reliability enhancement may severely affect a design’s cost and design timeline.

Ideally, reliability evaluation of a new design should be fast, accurate and available very early in the design lifecycle. But these attributes are often contradictory. Early models tend to be more abstract and with missing details and thus, less accurate. In addition, fast assessment methodologies tend to be inaccurate or very coarse-grain. This is why many reliability evaluation techniques exist and can be performed on different abstraction levels, with each combination offering different tradeoffs between speed and accuracy. Reliability evaluation techniques can be summarized as Statistical Fault Injection (SFI) [5] [6] [7], Architecturally Correct Execution (ACE) [8] [9] [3] analysis and analytical methods [10] [11] [12]. SFI is often the most time consuming, as it requires multiple simulations in order to gather a statistically significant sample. However, it is very accurate and provides insights on the fault effects, as it experimentally determines the vulnerability of a system and the fine-grain effect of every injected fault. ACE analysis, on the other hand, is a technique that aims to profile the data lifetime inside the hardware structures and quantify their exposure, to estimate the vulnerability. It is significantly faster, as it only requires a single or few simulations, but requires a lot of development effort (due to the complexity of tracking down data on a complex microprocessor) and does not provide fine-grain insights on the fault effects. Finally, statistical methodologies provide estimations on the reliability based on statistical events that correspond to how each hardware structure was utilized. While it is the fastest method and little to no development effort, it is also the most inaccurate among all [6].

In this work, we present *reverse ACE* (rACE), a new tracing technique that aims to improve the accuracy of ACE analysis. rACE is applied on the execution trace of a program and builds dependency trees of instructions that are associated with the data output and the control flow. Our proposed methodology improves the accuracy of ACE analysis while it also provides detailed fault-effect classification. We have implemented rACE on top of Gem5, a microarchitecture-level simulator and evaluated its accuracy and granularity of analysis against traditional ACE analysis and Statistical Fault Injection.

II. Methodology

This section initially summarizes the basics of ACE analysis and implementation improvements from the literature, before presenting reverse ACE and its implementation on the physical register file of an out-of-order CPU core.

A. ACE Analysis

Architectural Correct Execution (ACE) analysis is a vulnerability estimation methodology that relies on tracking whether the data of an entry is eventually affecting the visible program output. Unlike fault injection, which experimentally evaluates the ACEness of one bit per run, ACE analysis calculates the AVF of a component on a single simulation run. The performance benefits of the analysis are obvious, compared to fault injection techniques that suffer from the very long estimation times. However, ACE analysis also comes with some significant drawbacks: (a) it reports overly pessimistic (high) vulnerability figures for the microprocessor components and (b) it does not deliver any fine-grain information about the effects of faults during program execution.

ACE analysis is performed at the granularity of data entries. This means that a bit or set of bits is expected to have the same ACEness between two accesses, no matter of the distance between the two events. For instance, if a 32-bit register is stored in clock cycle 100 and then it is read in clock cycle 150, this means that the ACEness of all bits between clock cycle 100 and 150 will be the same. Faults that strike in clock cycle 110 and 123 are expected to have the same effect on the program, as they are both "read" on clock cycle 150 and in the same manner.

Once a data entry is accessed, it has to be tracked until it, directly or indirectly, reaches (or not) the final program output. Unfortunately, this process is really complex as it involves all software and hardware masking effects. For instance, a data entry that is read and used in a condition check that will drive a branch can change the program flow and eventually affect the program output. If the condition is, for instance, a comparison with an immediate value (e.g. termination condition of a for loop), then a fault in the data entry will always cause a different amount of iterations in the loop. But if the comparison is a "not zero" if statement, then, unless the value is typically zero at that point, the program will remain unaffected, even if the entry is accessed and participates in a control instruction. On the other hand, a read in the hardware level can be associated with a speculatively executed operation, which can initiate a memory request that may evict a cache line. Depending on the current hardware state, the line may be marked as "owned" in the cache line and thus, eviction will keep a copy of the faulty entry in the memory hierarchy. In the opposite case where a copy already exists in the next memory level and there is no ownership, the cache line will be dropped, discarding any faults in that particular entry.

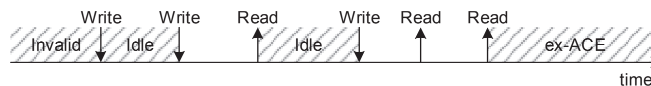

Fig. 1: Example entry accesses on hardware level. Periods before a write operation are considered Idle or Invalid, and are always un-ACE; only white-spaced intervals are ACE.

It is practically impossible to know what will happen and how the entry is going to be used, especially in complex systems and programs. This is the reason why ACE analysis is reported to suffer by significant levels of inaccuracy (overestimated vulnerability of structures) [13] [14]. Many works have proposed techniques on performing ACE analysis [15] [9] [3] [16] and methods to reduce part of the inaccuracy. These can be summarized as (according to [3]):

For the microarchitecture level:

- Idle or Invalid State: Data and status bits that are idle or do not contain valid information are un-ACE bits.
- Mis-speculated State: The bits that represent incorrectly speculated operations are un-ACE bits.
- Performance Structures: All speculative structures (e.g. branch predictors) contain only un-ACE bits.
- Ex-ACE State: ACE bits become un-ACE bits after their last use. This category encompasses both architecturally dead values, such as those in registers, as well as architecturally invisible state.
- Y-Bits: Dynamic bits of state that have the power to dramatically alter the course of execution, but eventually do not affect it.

Fig. 1 illustrates an example series of accesses upon a hardware data entry, and how idle, invalid and ex-ACE states are formed. Every write action immediately discards whatever was stored in the entry and marks an un-ACE state for the entry until the previous most recent read. If there was no prior read, the state is marked as invalid. Similarly, the last ACE entry state is marked by a Read access. After that, the state becomes ex-ACE.

For the architecture level:

- NOP instructions: The bits of a NOP instruction (apart from the opcode) are un-ACE bits.
- Performance-enhancing instructions: A single-bit upset in a non-opcode field of e.g. a prefetch instruction will not affect the correct execution of a program.
- Dynamically dead instructions: Dynamically dead (DD) instructions are those whose results are not used. Instructions whose results are simply not read by any other instructions. Tracking for DD instructions is often performed on a finite instruction window.
- Logical masking: There are many bits that belong to operands in a chain of computation whose values still do not

```
14004872818311000: system.cpu T0 : 0x8f90    :   b                       : IntAlu :
14004872818316500: system.cpu T0 : 0x8f6a    :   mov   r2, r4            : IntAlu :  D=0x0000000000000000
14004872818316500: system.cpu T0 : 0x8f6c    :   movs   r3, r4, ASR #31  : IntAlu :  D=0x0000000000000001
14004872818317000: system.cpu T0 : 0x8f6e    :   str   r6, [sp, #8]      : MemWrite :  D=0x0000000000004dec4 A=0x7efbf4f8
14004872818317500: system.cpu T0 : 0x8f70    :   add   r0, sp, #20       : IntAlu :  D=0x000000007efbf504
14004872818318000: system.cpu T0 : 0x8f72    :   strd.w  r2, r3, [sp, #0] : MemWrite :  D=0x0000000000000000 A=0x7efbf4f0
14004872818318000: system.cpu T0 : 0x8f76    :   mov   r2, r1            : IntAlu :  D=0x0000000000000c7d
14004872818318000: system.cpu T0 : 0x8f78    :   movs   r3, r1, ASR #31  : IntAlu :  D=0x0000000000000001
14004872818318000: system.cpu T0 : 0x8f7a    :   add   r4, r4, r1        : IntAlu :  D=0x0000000000000c7d
14004872818318500: system.cpu T0 : 0x8f7c    :   bl.w                    : IntAlu :  D=0x0000000000008f81
14004872818331000: global: PseudoInst::writefile(0x7efbf504, 0xc7d, 0x0, 0x4dec4)
```

Fig. 2: Reverse ACE dependency resolution. Writefile() instruction indicates the position of the output. By moving backwards in the trace, we can locate the last instruction (red) that stored in the output region. The instruction has two dependencies, which are set by the green instructions. The sequence continues until all of the instruction tree is resolved.

influence the computation's results. These bits are considered logically masked.

B. Implementation

We used Gem5 simulator [17] to implement the existing proposed ACE analysis for the physical register file of an out-of-order core running ARM and x86 binaries for all experiments. We have enhanced Gem5 to enable tracking of all of the described microarchitecture level situations that define the ACEness of an entry, including discarding the entries associated with mis-speculated instructions. These are integrated in the baseline form of ACE analysis that our setup can perform. This mode is referred as *baseline ACE*.

On the architecture-side, we have added two additional modes to account for the described architecture conditions. The first is a new detailed approach for tracking logical masking. When logical masking resolution is enabled, our modified Gem5 includes an additional step in the execution of every instruction. It temporarily stores the execution outcome of the instruction and then re-executes it with a fault on every bit of its operands to determine if the fault leads to a different result. This extra feature resembles Statistical Fault Injection, but unlike it, it exhaustively evaluates every single instruction, but only in the context of the functional unit. This approach additionally resolves Y-Bits, as proposed in [18]. The required overhead is relative to the number of bits of the operands. For instance, on a 32-bit ISA, a typical instruction has two operands, which means that every instruction will be re-executed 64 times. As only a small part of the simulation is actually repeated, the time required for it is always less than the time of 64 simulations, which is a very reasonable simulation time target, especially when considering that fault-injection runs are often in the scale of 1000x. The logical masking resolution overhead is typically less than 10x of a single-simulation time. Existing literature does not propose alternative implementation approaches to cover logical masking and thus, the required time for the proposed improvements can be variable and go beyond the single-execution concept of ACE analysis.

The second architecture-level feature for the refinement of ACE is the resolution of dynamically dead (DD) instructions. This also covers NOP and performance-enhancing instructions under the same implementation. Our implementation allows tracking of DD instruction in a given instruction window which can be even the entire program. To do so our implementation requires to perform an additional profiling simulation that will generate a program flow trace, on top of which it will perform the dependency analysis. As a result, the required time is 2x on the single simulation. The two architectural features are optional and can be enabled together or individually on our Gem5 implementation. In our setup we use the term *enhanced ACE* for the setup that includes both the architecture-level features. Notice that for the DD instruction tracking we use an infinite window, i.e. till the end of the program execution.

C. Reverse-ACE Analysis

ACE analysis overestimates vulnerability, as a result of its pessimistic nature and the high complexity in tracking down the lifetime of data and how they can affect the program output. The existing architecture-level techniques are very limited and only track down whether the outputs of an instruction are actually used by other instructions or not. While this technique can filter out instructions that are certainly not affecting the program (and thus the output), it is not correlating at all with the actual program output due to the difficulty of keeping track of all the instructions.

To address this issue, we have developed the *reverse ACE analysis* (rACE). Instead of tracing instructions in the program flow to see if their results are used in any way, in reverse ACE, we start from the program output and moving backwards in the program flow, we resolve all of its dependencies. The output in Gem5 simulation is placed on the system's memory and exported to the host system for comparison through a "m5_writefile()" pseudo-op. Using the execution trace file and by parsing the arguments of the pseudo-op, we are able to locate the memory addresses that contain the output, and this defines the starting point of the reverse ACE.

Having the list of memory addresses that contain the program output, we move backwards to the instruction trace in order to find the instructions that did the latest stores in these addresses. These will contain new dependences on what was stored there, and following the same trace, we locate the instructions that set the dependencies. Eventually, this results to an instruction tree that is somehow correlated with the actual program output. Following up, the ACE analysis is then performed in the entries that are correlated with this instruction tree, instead of the whole program. Fig. 2 illustrates how the resolution looks like in a Gem5 instruction trace. Notice that for the implementation of reverse ACE, our Gem5 implementation generates a custom trace file, similar to the one shown in the figure that includes the instruction ID (unique per instruction in

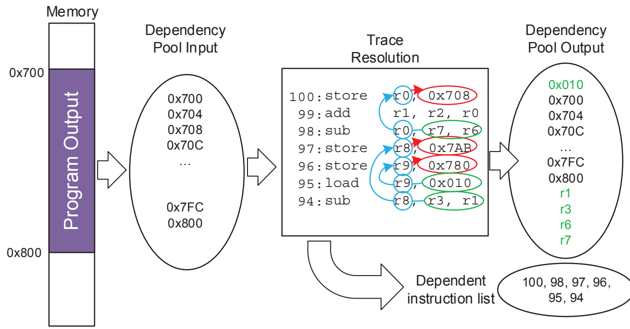

Fig. 4: Reverse ACE analysis flow.

the simulation), the list of sources of the instruction and list of destinations. Redundant information such as data transactions, disassembled instruction format, execution time, etc. have been removed to speed up the analysis and to reduce the size of the tracefile. In addition, the customized tracefile is built on top of the out-of-order core and is universal across different ISAs.

The reverse ACE analyzer component has two inputs: a trace file to be resolved and an initial dependency pool. The pool contains all of the current "active" dependencies that need to be resolved. For each instruction, and in the reverse order (output to input), the list of destinations is searched to find whether any of the active dependencies is set by the instruction. If a new dependency is found, the instruction is marked as dependent, the dependency is removed from the pool and the sources of the instruction are added to the pool. The same sequence continues until the end (i.e. since we are in reverse order, until the beginning) of the trace file is reached. Once the entire trace file has been processed, the instruction list and current (unresolved) dependencies in the pool are stored as output files. The flow is illustrated in Fig. 3. The instruction list is then passed to Gem5 to guide the ACE analysis on using only data-related instructions. Similarly, the tool can be also configured to consider *all control instructions* (which can also cause control failures such as crashes and hangs) as ACE and track their dependencies in a similar manner. This way, rACE analysis can extract two instruction trees, the *data-related tree* and the *control-related tree*. Instructions that belong to both trees are attributed to the vulnerability of both trees. Having two separate instruction trees, the ACE analysis can provide an estimation that is related to the tree, and associate it with a separate fault effect: data-related tree can provide a Silent Data Corruptions (SDC) estimation, while control-related tree can provide a Crash estimation (or Detected Unrecoverable – DUE estimation).

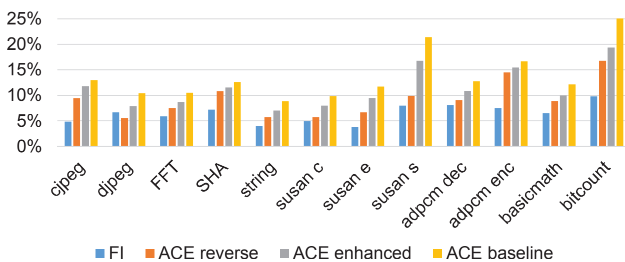

Fig. 3: AVF estimation using Fault injection, Reverse ACE, Enhanced ACE and baseline ACE analysis for a register file of 128 entries using ARM ISA.

III. Experimental Results

We have enhanced Gem5 simulator with the capability of ACE and rACE analysis on the physical register file of the out-of-order CPU core. The same methodology can be applied to all hardware structures that are directly associated with executed instructions, such as reorder buffer, load-store queue etc. Using three differently sized register files of 64, 96 and 128 physical registers, we have assessed the vulnerability of 12 benchmarks of the MiBench [19] suite, on x86 and ARM architectures.

A. AVF estimation

We compare baseline ACE, enhanced ACE and reverse ACE estimation against fault-injection. For the fault-injection experiments, we used GeFIN fault injection framework [20] on the same hardware and ISA configurations and using a sample of 2000 injections which corresponds to statistically significant campaign of only 1.8% error margin and high confidence of 99% [5]. Fig. 4 presents the estimation of each configuration.

Consistently, for all benchmarks, Reverse ACE reports an narrower AVF estimation than baseline and enhanced ACE. Notice that the reported enhanced ACE is actually more accurate than traditional ACE, as it can track dynamically dead instructions in the whole program without limits. It also includes logical masking resolution. The baseline ACE includes resolution of all microarchitecture level states, including idle or invalid, ex-ACE mispeculated and performance (which in the case of register file is not applicable).

When compared to Fault Injection, which in our case is considered the reference point in terms of accuracy, baseline ACE is reported to overestimate from 2X to 3X. Enhanced ACE reports an overestimation from 1.4X to 2.5X (average 1.8X), while reverse ACE has the best accuracy among *all* three and for *all* benchmarks with 1.45X average overestimation. The reported differences are significantly less compared to existing reports of 3X and 7X of the literature [13] [14] [18], which however were not performed on the exact same models, unlike our case. Interestingly, we can observe that for djpeg program, reverse ACE appears to underestimate the vulnerability. The exclusion of instructions that do not participate on the data-related or control instruction tree (e.g. a dead instruction that performs a division by zero will raise an exception) are factors that cause potential underestimation of the results, and this is the case for djpeg benchmark.

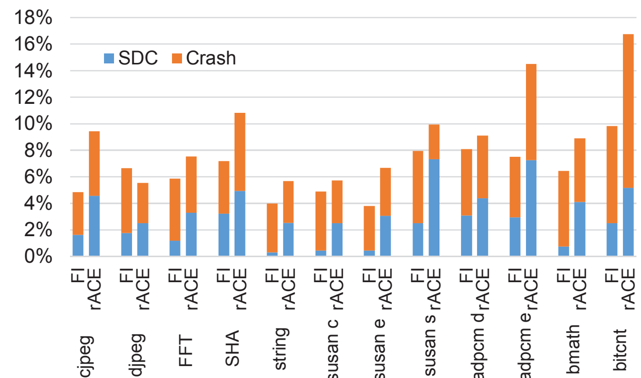

Fig. 5: AVF estimation of SDCs and Crashses for rACE and FI.

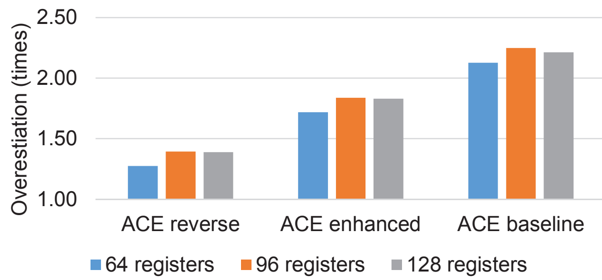

Fig. 6: Overestimation of the three ACE methodologies compared to Fault Injection for three different register file sized.

B. *Fine-grain fault effect classification*

rACE contributes also in improving the second drawback of ACE analysis: the fault effect classification granularity. Using reverse ACE, we can distinguish two instruction trees from the program flow, one that is associated with the program output and one that is associated with the control flow. We can get separate estimations using these two trees, providing an indirect way of estimating output and control vulnerability. Instructions that belong to the data-related tree can cause corruptions to the program output (SDC), while instructions that belong to the control-tree can cause program or system crashes (Crash, DUE).

As already mentioned, instructions from the dynamically dead tree or the data-related tree can also cause exceptions or segmentation faults, which result to crashes, but this is a limitation that already exists in ACE analysis and is considered as source of underestimation. Fault injection experiments on the other hand offer high level of observability and fine fault effect classification. Fig. 5 shows the results of the comparison. In order to have a common ground for the comparison of the two, we group all fault effects of the FI that correspond to malfunctions, crashes and timeouts to a single class, stated as *crash* which corresponds to the estimation on the control-tree of rACE (which is also stated as *crash*). The Silent Data Corruption class of FI directly corresponds to the data-related tree of rACE, which we refer as *SDC* in both cases.

C. *Different Register File Sizes*

ACE analysis focuses on quantifying for how long useable entries remain exposed on a hardware structure, compared to the total exposure of the structure. When a structure size changes, it proportionally changes the total exposure time but not necessarily the exposure time of useable entries, especially if the running workload is not stressing the component. In order to observe how rACE compares to the other ACE implementations, we performed an estimation on physical register files with 64, 96 and 128 registers.

Fig. 6 shows the average overestimation of both x86 and ARM, for each of the ACE methodologies against Fault Injection and for three different sizes on the register file. All methodologies appear to have steady level of overestimation for the different structure sizes, with the 64 register size being slightly more accurate than the other two. This trend is observed in all cases. rACE provides the tightest AVF measurements in all cases with only 1X to 1.8X difference compared to FI.

IV. Related Work & Other uses

ACE analysis was introduced by Mukherjee *et.al.* [3], along with the definition of Architectural Vulnerability Factor. Biswas *et.al.* [8] extended the original ACE analysis with several optimizations, most of which concerned SRAM based structures. Wang *et.al.* [18] quantified the level of inaccuracy of ACE analysis by comparing against fault injection. They also proposed new optimizations for the analysis to improve the accuracy. Biswas *et.al.* elaborated on how additional development efforts can further improve the analysis [15]. Other works have also compared ACE analysis against fault injection to quantify the level of overestimation [13] [14] [21]. In this work, we further extend the accuracy of ACE analysis with the reverse dependency analysis and the logical masking implementation.

Reliability estimation using ACE analysis has been extensively used in the literature [11] [22]. Sim-Soda [9] is a simulator that natively supports ACE analysis. Authors of [16] use ACE analysis to evaluate a newly proposed statistical estimation model. Sridharan *et.al.* used ACE analysis to demonstrate how the vulnerability of a system can be evaluated in separate layers [23] [24]. Additionally, they extended the ACE analysis in order to additionally estimate spatial multi-bit fault vulnerability [25]. Authors of [21] and [26] employed ACE analysis to profile and reduce the number of fault injections for a particular sample. Statistical fault injection has also been extensively used in the past for reliability estimation on different abstraction levels. Microarchitecture level fault injection has been used in [7] [27] [20] [6], RTL fault injection [28] [29] [30] and architecture (software) level [31] [32] [33]. Chatzidimitriou *et.al.* [34] have used neutron beaming against microarchitecture-level fault injection to quantify the accuracy of the latter.

A. *Other applications of rACE*

The rACE analyzer can perform dependency resolution analysis on traces that are split to smaller chunks. This ability can be used to enhance fault-injection experiments that typically require complete simulations in order to determine whether a fault has affected the output. Using reverse ACE, a fault injection framework can determine the vulnerability of smaller portions of a workload. The analyzer can trace back the dependency tree, starting from the output file and stopping at the end of the desired program portion. By knowing which the active dependencies on the dependency pool at the end of the program part are, a fault injection campaign can be used to attribute the mismatches to the output dependent instructions.

V. Conclusion

We have proposed reverse ACE (rACE) a novel, reverse-order dependency resolution technique that is a major enhancement of traditional ACE analysis. rACE provides finer means of dynamically dead instruction detection concept, that allows distinction of SDC and Crash AVF. We have evaluated the methodology on top of the state-of-the-art microarchitectural simulator Gem5 and compared the estimation against fault injection. The reverse ACE appears to overestimate AVF by 1.45X, when baseline ACE is measured to overestimate by 2.3X on average and the enhanced ACE by 1.8X.

REFERENCES

[1] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers,* vol. 22, pp. 258-266, 5 2005.

[2] C. Constantinescu, "Intermittent Faults and Effects on Reliability of Integrated Circuits," in *Reliability and Maintainability Symposium (RAMS)*, Las Vegas, NV, USA, 2008.

[3] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Washington, DC, USA, 2003.

[4] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[5] R. Leveugle, A. Calvez, P. Maistri and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, Nice, France, 2009.

[6] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris and D. Gizopoulos, "Differential Fault Injection on Microarchitectural Simulators," in *2015 IEEE International Symposium on Workload Characterization*, 2015.

[7] G. Yalcin, O. S. Unsal, A. Cristal and M. Valero, "FIMSIM: A fault injection infrastructure for microarchitectural simulators," in *IEEE International Conference on Computer Design (ICCD)*, Amherst, MA, USA, 2011.

[8] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee and R. Mukherjee, "Computing Architectural Vulnerability Factors for Address-Based Structures," in *International Symposium on Computer Architecture (ISCA)*, 2005.

[9] X. Fu, T. Li and J. A. B. Fortes, *Sim-SODA: A Unified Framework for Architectural Level Software Reliability Analysis.*

[10] L. Duan, B. Li and L. Peng, "Versatile prediction and fast estimation of Architectural Vulnerability Factor from processor performance metrics," in *International Symposium on High Performance Computer Architecture (HPCA)*, Raleigh, NC, USA, 2009.

[11] G.-H. Asadi and others, "Balancing Performance and Reliability in the Memory Hierarchy," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, Washington,DC,USA, 2005.

[12] A. Vallero, A. Savino, G. Politano, S. D. Carlo, A. Chatzidimitriou, S. Tselonis, M. Kaliorakis, D. Gizopoulos, M. Riera, R. Canal, A. Gonzalez, M. Kooli, A. Bosio and G. D. Natale, "Cross-layer system reliability assessment framework for hardware faults," in *2016 IEEE International Test Conference (ITC)*, 2016.

[13] N. George, C. R. Elks, B. W. Johnson and J. Lach, "Transient fault models and AVF estimation revisited," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, Chicago, IL, USA, 2010.

[14] X. Li, S. V. Adve, P. Bose and J. A. Rivers, "Architecture-Level Soft Error Analysis: Examining the Limits of Common Assumptions," in *EEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Edinburgh, UK, 2007.

[15] A. Biswas, P. Racunas, J. Emer and S. S. Mukherjee, "Computing Accurate AVFs using ACE Analysis on Performance Models: A Rebuttal," *IEEE Computer Architecture Letters,* vol. 7, pp. 21-24, 1 2008.

[16] A. A. Nair, S. Eyerman, L. Eeckhout and L. K. John, "A first-order mechanistic model for architectural vulnerability factor," in *International Symposium on Computer Architecture (ISCA)*, Portland, OR, USA, 2012.

[17] 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, pp. 1-7, 8 2011.

[18] N. J. Wang, A. Mahesri and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," *ACM SIGARCH Computer Architecture News,* vol. 35, p. 460, 6 2007.

[19] 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 *IEEE International Workshop on Workload Characterization (WWC)*, 2001.

[20] A. Chatzidimitriou and D. Gizopoulos, "Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Uppsala, Sweden, 2016.

[21] M. Kaliorakis, D. Gizopoulos, R. Canal and A. Gonzalez, "MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[22] A. A. Nair, L. K. John and L. Eeckhout, "AVF Stressmark: Towards an Automated Methodology for Bounding the Worst-Case Vulnerability to Soft Errors," in *IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, USA, 2010.

[23] V. Sridharan and D. R. Kaeli, "Using Hardware Vulnerability Factors to Enhance AVF Analysis," in *International Symposium on Computer Architecture (ISCA)*, New York, NY, USA, 2010.

[24] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Raleigh, NC, USA, 2009.

[25] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi and D. R. Kaeli, "Calculating Architectural Vulnerability Factors for Spatial Multi-Bit Transient Faults," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, 2014.

[26] X. Xu and M.-L. Li, "Understanding soft error propagation," in *IEEE/IFIP International Conference on Dependable Systems and Networkds (DSN)*, 2012.

[27] 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)*, Asheville, NC, USA, 2013.

[28] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proceedings of the 50th Annual Design Automation Conference on - DAC 13*, 2013.

[29] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas and Y. Makris, "Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller," *IEEE Transactions on Computers,* vol. 60, pp. 1260-1273, 9 2011.

[30] A. Chatzidimitriou, M. Kaliorakis, D. Gizopoulos, M. Iacaruso, M. Pipponzi, R. Mariani and S. D. Carlo, "RT Level vs. Microarchitecture-Level Reliability Assessment: Case Study on ARM(R) Cortex(R)-A9 CPU," in *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Denver,CO,USA, 2017.

[31] R. Natella, D. Cotroneo, J. A. Duraes and H. S. Madeira, "On Fault Representativeness of Software Fault Injection," *IEEE Transactions on Software Engineering,* vol. 39, no. 1, pp. 80 - 96, 2013.

[32] J. Wei, A. Thomas, G. Li and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[33] Q. Lu, M. Farahani, J. Wei, A. Thomas and K. Pattabiraman, "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015.

[34] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos and P. Rech, "Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Portland, OR, USA, 2019.