PRESAGE: Selective Low-Overhead Error Amplification for Easy Detection

Vishal C. Sharma
University of Utah
Salt Lake City, UT
vishalcsharma.cse@gmail.com

Arnab Das
University of Utah
Salt Lake City, UT
arnabd@cs.utah.edu

Ian Briggs
University of Utah
Salt Lake City, UT
ianbriggsutah@gmail.com

Ganesh Gopalakrishnan
University of Utah
Salt Lake City, UT
ganesh@cs.utah.edu

Sriram Krishnamoorthy
Pacific Northwest National
Laboratory
Richland, WA
sriram@pnnl.gov

KEYWORDS

soft error detection, loop programs, address generation

ACM Reference format:
Vishal C. Sharma, Arnab Das, Ian Briggs, Ganesh Gopalakrishnan, and Sriram Krishnamoorthy. 2017. PRESAGE: Selective Low-Overhead Error Amplification for Easy Detection. In *Proceedings of ACM Supercomputing conference, Denver, Colorado USA, November 2017 (SC'17),* 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Virtually all system resilience solutions to protect applications against silent data corruptions involve several key subsystems including error detectors and recovery mechanisms. Unfortunately, the design of efficient data-space detectors has proven to be particularly difficult. In the literature, one finds detectors based on time-series data analysis [4], special detectors for load/store integrity checking [8] as well as data outlier detectors that are constructed based on learning an approximated version of the intended computation [6].

The real challenge with error detectors is their non-trivial runtime overheads running into 50% or more. Given this high overhead, one expects that the detectors produce virtually no false positives and miss very few errors. Unfortunately, the false positive rates either tend to exceed the inherent bit-flip propensity itself—or if the detection rate is tuned to be very low, the omission rate tends to grow, turning the detectors into a net source of runtime slowdown, and nothing more. Unfortunately, these inconvenient truths tend to get lost in the midst of successes in other areas, including good recovery mechanisms—but without good detectors, nothing else matters that much.

Let us consider an alternative approach in which computations are cut up into sequential chunks of activities where each activity is allowed to run nearly full-speed. If the activity succeeds, the computation gainfully moves onto the next activity. However if a bit flip occurs amidst one activity, we *amplify the fault* so that the fault is made to cascade through the rest of the activity, thereby *facilitating an easier detection* of the fault at the end of the activity.

SC'17, November 2017, Denver, Colorado USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

While a detector is still involved, the key difference is that in the speed-critical parts of the code (e.g., loops), the emphasis is not on detection but *fault amplification*. Detection is postponed to loop exit points where detection occurs with very high certainty and far lower overall overhead. Is this kind of fault amplification even possible? Our work provides one example of this idea being realized and extensively evaluated.

2 FAULT AMPLIFICATION BY ADDRESS RELATIVIZATION

With the guiding principle of divide and conquer as one of the most suited strategies to improve system resilience, we focus on making *structured address computation more robust*. These occur in many codes such as stencils where an offset expression is added to a fixed base address to form an effective address. The base remains the same while the offset keeps incrementing according to some logic. If one offset expression calculation is hit by a bit flip, the effective address may stray out of bounds, and the resulting crash is indeed "detection." However, in today's large address spaces, instead of such crashes, one ends up injecting a completely irrelevant piece of data into the calculation for *just one computational step*—but then future effective addresses may most likely be correct. Such errors will prove exceedingly hard to detect, but may build up over time in the data space. Our thrust is to instead let the fault persist in the effective-address space, as we may then cheaply detect the fault as an incorrect address emerging out of a loop.

3 IMPLEMENTATION: LLVM TRANSFORMATION

Our PRESAGE [7] approach rewrites LLVM code so that every fixed base plus offset address calculation is turned into a moving base plus a delta. The moving base is the previous relative address calculated. Thus, if one relative address faults, all following relative addresses also fault. In the absence of the fault, the code behaves functionally the same. The LLVM transformations are made pervasive end-to-end in the code; thus, one fault at the beginning cascades till the end.

Four important questions now are: (1) how much slowdown? (2) does relativization survive memory-oriented compiler transformations? (3) do some ISAs help reduce overheads? (4) what

detection rate can be achieved in real applications? We now present these results.

4 RESULTS

Our results indicate that the relative slowdown can be modest compared to many detection schemes: "20%" slowdown is achievable for benchmarks such as jacobi-2d and trmm running on an x86 platform. The detection rate for structured-address calculation soft errors is 100%. This combination of low overhead and high detection efficacy is not found in any conventional detection scheme we know of.

For the ARM ISA, we observe that one of its addressing modes allows the previous relative address to be cheaply reused for the next calculation. The pre-indexed and post-indexed addressing modes of the ARM ISA both modify the register holding the memory pointer. Since this modification is a design feature it does not have the same slowdowns associated with x86. Exploiting this code generation option has allowed us to reduce the overhead to zero in many cases. While found to exist serendipitously in ARM, this observation nevertheless has the potential to inform hardware manufacturers, encouraging them to add inexpensive future error detection features.

We performed extensive experiments using PLUTO [1–3] and found that the relativization is preserved across optimizations using polyhedral transforms under different tiling modes. Fig-4 in the poster suggests this finding, wherein the optimized code under PRESAGE transformation also shows higher crash (and thus detection) rates. Thus, PRESAGE can be combined with optimizations.

We now detail the fact that we obtain 100% Silent Data Corruption (SDC) detection *for structured address calculation*, plus other results. As Fig-3 of the poster suggests, PRESAGE shows an efficacy of 100% SDC detection for the given benchmarks. The unique design of the loop detector involves computing the exact address at the end of the dependency chain based on the base address and the original final offset. Once an error disturbs the dependency chain created for relativization, it is guaranteed to deviate from the exact final address at the detector (at the loop exit). Hence, even minor deviations are amenable to traps, which explains the high detection rate achieved by PRESAGE.

4.1 Result Breakdown

Although application of PRESAGE on *with* and *without* Pluto transformed code keeps the crash rate to be relatively unchanged. However, we do notice a balance shift between the SDC rates and BENIGN rates for these solvers under the transformations (Fig-4 of the poster). The LU example shows a significant increase in SDC rates even though its crash rate also increases under transformation. However, ADI indicates a fall in SDC rates under transformation. For example, in our experiments, baseline (with no transformations) LU has significantly low (0-2%) SDC rate which increases by around 5% under transformations, keeping the crash rates moderately unchanged. Unlike LU, ADI has high basline SDC rate of around 18%, which tends to drop by 5% on applying the transformations. We have run experiments on a wide variety of kernels from the Polybench benchmark suite [5]: LU, Jacobi, seidel, adi, mvt, dct, etc. Overall, we see a pattern wherein some of the stencil and solver codes are more prone to SDCs and transformations tend to enhance the SDC effect by promoting the otherwise BENIGN effects to SDCs, while some stencil and solver codes with higher baseline SDC rates tend to promote them to crashes under PRESAGE transformation. Hence, a follow-up investigation is required to investigate the susceptibility of specific instruction types in these codes to SDC or BENIGN effects to reason about the different trends observed in these experiments.

5 STRENGTHS, LIMITATIONS, AND FUTURE WORK

By focusing on one aspect (namely, structured address generation) of the whole resilience landscape, we are able to deliver strong results. The key limitation is, of course, that all other aspects of system resilience still need solutions. However, whatever these solutions are, they can be combined with PRESAGE.

We currently do not handle multi-dimensional arrays or arrays of structs. These features are currently being added to PRESAGE, as there is no fundamental reason (other than coding-time) why these features can't be entertained.

Another future work direction will be to look for further avenues for failure amplification, covering far more fault types than faults affecting structured address calculations. Also, "faults in structured addresses" is a fairly high-level concept in that by detecting these faults, it is unclear how they correspond with either binary instruction-level faults or even register-transfer level faults. Gaining this understanding would add significant insights to an overall effective cross-layer resilience solution.

REFERENCES

[1] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. 2008. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *International Conference on Compiler Construction (ETAPS CC)*.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A Practical Automatic Polyhedral Program Optimization System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[3] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. 2007. *PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer*. Technical Report OSU-CISRC-10/07-TR70. The Ohio State University.

[4] Sheng Di and Franck Cappello. 2015. Adaptive Impact-Driven Detection of Silent Data Corruption for HPC Applications. *IEEE Transactions on Parallel and Distributed Systems* (12/2015 2015). https://doi.org/10.1109/TPDS.2016.2517639

[5] polybench [n. d.]. PolyBench/C: The Polyhedral Benchmark suite. http://web.cs.ucla.edu/~pouchet/software/polybench/. ([n. d.]).

[6] V. Sharma, G. Gopalakrishnan, and G. Bronevetsky. 2015. Detecting Soft Errors in Stencil Based Computations. In *11th workshop on Silicon Errors in Logic - System Effects(SELSE)*.

[7] Vishal Chandra Sharma, Ganesh Gopalakrishnan, and Sriram Krishnamoorthy. 2016. PRESAGE: Protecting Structured Address Generation against Soft Errors. In *23rd IEEE International Conference on High Performance Computing, HiPC 2016, Hyderabad, India, December 19-22, 2016*. IEEE Computer Society, 252–261. https://doi.org/10.1109/HiPC.2016.037

[8] Sanket Tavarageri, Sriram Krishnamoorthy, and P. Sadayappan. 2014. Compiler-assisted detection of transient memory errors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 204–215. https://doi.org/10.1145/2594291.2594298