Parallelizing Software-Implemented Error Detection

Ute Schiffel, André Schmitt, Martin Süßkraut,
Stefan Weigert, and Christof Fetzer

Technische Universtät Dresden
Department of Computer Science
http://wwwse.inf.tu-dresden.de
Dresden, Germany
{ute, andre, suesskraut, stefan, christof}@se.inf.tu-dresden.de

(Research paper)

Abstract. Because of economic pressure, more commodity hardware with insufficient error detection is used in critical applications. Moreover, it is expected that commodity hardware is becoming less reliable because of the continuously decreasing feature size. Thus, we expect that software-implemented approaches to deal with unreliable hardware will be needed. Arithmetic codes are well suited for this purpose because they can provide very good error detection capabilities independent of the actual failure modes of the underlying hardware. But arithmetic codes generate high slowdowns. This paper describes our encoding which uses an expensive AN-code. Second, we show how we harness the power of modern multicore CPUs to parallelize this expensive but flexible and powerful software-implemented fault detection technique. Our measurements show that under continuous probabilistic error injection, AN-encoding reduces the number of runs with incorrect output from 15.9% for the unencoded execution to 0.5% in the encoded case. Our parallelization reduces the observed slowdowns by an order of magnitude.

1 Introduction

Historically, hardware reliability has been increasing with every new generation. In the future, it is expected that decreasing feature size of hardware will, however, lead to less reliable hardware [5]. Moreover, the error rate in logical circuits has overtaken error rates in memory [8]. Thus, the usage of memory protection alone is not sufficient anymore.

Historically, critical and especially safety-critical systems have mostly been built using special purpose hardware with better error detection and masking with the help of redundancy. Some hardware is even radiation hardened to prevent environment induced execution errors. However, these solutions are expensive and usually an order of magnitude slower than commodity hardware. We expect that in the future there will be even a higher economic pressure to use commodity hardware for dependable computing. Therefore, there is a need to

cope with the restrictive failure detection capabilities of commodity hardware in software. Commodity hardware will not exhibit pure fail-stop behavior but instead exhibit value failures which are much more difficult to detect and to mask. We aim at implementing a system which will turn these arbitrary value failures into easier to handle crash failures without the need for special hardware.

Arithmetic codes (see Sec. 2) facilitate software-implemented hardware error detection. In this paper, we use an *AN-code*. Its main advantage of arithmetic codes is that one can ensure error detection with a given probability - independent of the underlying hardware. Arithmetic codes introduce a very high overhead. Previous approaches [6, 12] reduced the overheads of arithmetic codes by not completely encoding applications and additionally, by using less powerful AN-codes. We, instead, protect every instruction of a program using the same powerful AN-code.

Section 3 demonstrates how we reduce the slowdowns of the encoding by parallelizing the encoded execution using the power of modern multicore CPUs. The measurements presented in Sec. 4 show that under continuous probabilistic error injection, AN-encoding reduces the number of runs with incorrect output from 15.9% for the unencoded execution to 0.5% in the encoded case. Note that one can improve the strength of the encoding (by using a larger A - see Section 2) to reduce the percentage of incorrect correct outputs even further. Of course, a larger A also increases the overheads. Our parallelization reduces the observed slowdowns by an order of magnitude on a 16-core system. We discuss the related in Sec. 5.

2 AN-Encoding

Arithmetic codes are a technique to detect hardware errors during runtime. The encoding adds redundancy to all data words. This results in a larger domain for data words and only a small subset of the domain contains the valid code words. Arithmetic codes are conserved by correctly executed arithmetic operations: a correctly executed operation given valid code words as input, outputs a valid code word. A faulty arithmetic operation destroys the code with a very high probability, i.e., results in an invalid code word [2].

In the following, we will briefly summarize our previous work with AN-code which is published in [14]. We want to give the reader a general idea about the concept and an understanding why the application of AN-code is as computationally expensive as it is.

The *AN-code* is one of the most widely known arithmetic codes. The encoded version x_c of variable `x` is obtained by multiplying its original functional value x_f with a constant A. This encoding is only done for input values of a program. All computations take multiples of A as inputs and if executed error-free, produce multiples of A as outputs. Code checking is done by computing the modulus with A, which is zero for a valid code word. Before a variable is externalized, i.e., used as a parameter of an external function or as a memory address in a

load or a store operation, it is checked if it is a valid code word. If the check fails, the application is aborted.

The advantage of an AN-code is that the probability of detecting an operation error does not depend on the used hardware but only on the choice of A. Assuming a failure model with equally distributed bit flips and a constant Hamming distance between all code words, the resulting probability of detecting one error is approximately: $1 - 2^{-k}$, where, k is the size of A in bits. Thus, A should be as large as possible. Furthermore, it should not be a power of two because a multiplication by A would only shift the bits to the left and no bit-flips in the higher bits would be detected. A should also have as few factors as possible to increase the probability of detecting an error. Large prime numbers are therefore a good choice for A.

For encoding a program with an AN-code, every variable has to be replaced with its larger AN-encoded version. Every instruction is substituted by its AN-preserving counter-part. We perform the instrumentation at compilation time:

- The scope of the protection includes compiler errors. For example, errors in lowering the source code to an executable binary will most likely result in invalid codes and hence, become detectable.
- We do not introduce further slowdowns because of dynamic instrumentation. See [21] for a detailed discussion of advantages and disadvantages of encoding at compile vs at runtime.

We have implemented our *encoding compiler* using the LLVM compiler framework [10]. We encode LLVM's bitcode which is a static single assignment assembler-like language. The advantage of LLVM's bitcode, in comparison to any native assembler, is its manageable amount of instructions for which we have to provide encoded versions and the LLVM framework for analyzing and modifying LLVM bitcode. For AN-encoding LLVM bitcode, we solved the following problems:

- We need encoded versions of all operations supported by LLVM. Therefore, we provide a set of *basic hand-encoded arithmetic operations* and a set of encodable *replacement operations* which uses only the basic arithmetic operations.
- We have to handle calls to un-encoded external libraries.
- AN-codes are only applicable to integers. Thus, we encode floating point operations by replacing them with encodable software implementations which make only use of integers.
- We have to provide encoded versions of all constants and initialization values. LLVM enables us to find and modify all those initializations. Thus, we replace them with appropriate multiples of A.
- For the encoding of memory content, a specific word size had to be chosen: we chose 32 bit. We require the compiler to align all memory accesses to that word size because only whole encoded words can be read.

Basic Hand-Encoded Arithmetic Operations. Executing arithmetic operations on AN-encoded data mostly requires some corrections to obtain a correctly

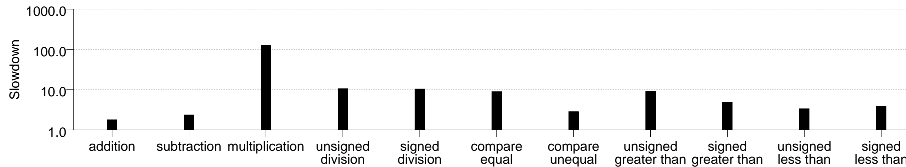

Fig. 1. Slowdowns of encoded arithmetic operations compared to their native versions.

encoded result. For example, consider the multiplication of two encoded values $a_c = A * a_f$ and $b_c = A * b_f$. When just multiplying a_c and b_c, the obtained result is $A^2 * a_f * b_f$ but the correctly encoded result should be $A * a_f * b_f$. Thus, an additional division by A is required. Our encoded arithmetic operations are hand-encoded. See [19] for the details of an AN-code with *signatures*, i.e., the value of a variable is not only multiplied by a factor A but also a unique constant is added.

Fig. 1 shows the slowdowns of our AN-encoded operations compared to their unencoded versions. While AN-encoded additions and subtractions take only two times as long as their native versions, an AN-encoded multiplication takes 126 times longer. Divisions and comparisons are between 3 and 10 times slower. The main reasons for those slowdowns are: (1) the implementation of the overflow behavior of integer operations as defined in the C standard, and (2) that encoded multiplications and divisions require expensive 128-bit integer operations.

Replacement Operations. Since encoding by hand is a tedious and error-prone task, we automated as much of the remaining encoding tasks as possible. Thus, we provide a library of so-called *replacement operations*. Those contain implementations of the following operations: shifts, casts, bitwise logical operations, and remainder operations. The replacement operations are written in such a way that they can be automatically encoded by our encoding compiler, i.e., they only use arithmetic operations for which hand-encoded versions exist. Before encoding a program, all not directly encodable operations, i.e., all operations which have no encoded variant in our basic set of hand-encoded *arithmetic operations*, are replaced with their appropriate encodable replacement operation.

Fig. 2 depicts the slowdowns generated by the slowest replacement operations compared to their native versions: for the unencoded and for the encoded replacement operations. Especially the bitwise logical operations (`not`x, `and`x, `or`x, and `xor`x) generate large slow downs. Their implementation uses tabulated results for 16-bit (`not`x) and 8-bit (the other operations) blocks and expensive shift operations to combine those results. Arithmetic right shifts (`ashr`x) are expensive because they require two accesses to tabulated data and an encoded division. Finally, the encoded versions of signed and unsigned remainder operations (`srem`x and `urem`x) and upcast and downcast operations (`sext-`x`-to-`y and `trunc-`x`-to-`y) still generate slowdowns between 8 and 64. They also require expensive encoded divisions.

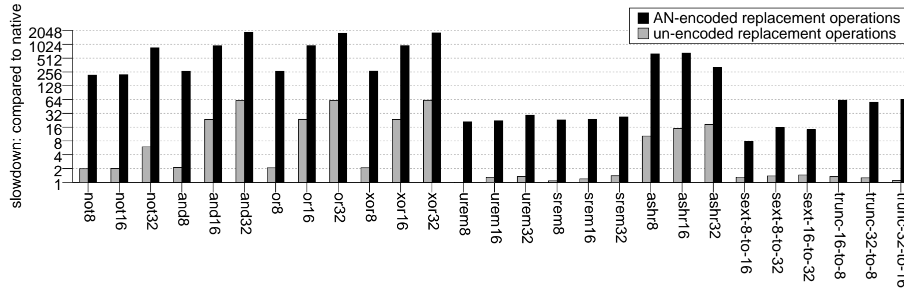

Fig. 2. Slowdowns of unencoded and AN-encoded versions of replacement operations compared to their native versions.

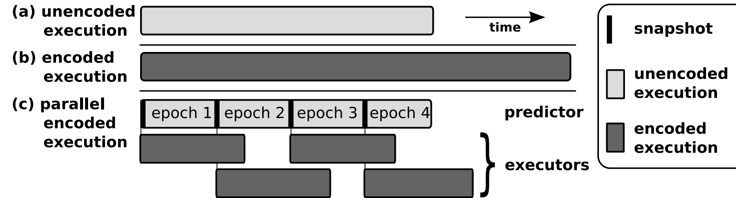

Fig. 3. (a) Original application execution without encoding and parallelization. (b) Sequential execution with encoding. (c) Parallelized encoded execution. Unencoded predictor for fast state prediction. Parallelized executors execute slow encoded variant.

Calls to unencoded External Code. In contrast to dynamic binary instrumentation, static instrumentation does not allow for protection of external libraries whose source code is not available at compilation time. For calls to those libraries, we currently provide hand-coded wrappers which decode parameters and after executing the unencoded original, encode the obtained results.

Note that AN-encoding leads to unexpected performance modifications. Some operations whose unencoded versions are very fast (casts, shifts, bitwise logical operations, multiplications and divisions) suddenly induce very large overheads. Depending on the encoded application AN-encoding results in slowdowns between 7.5 and 238 (see Sec. 4).

3 Parallelize Encoded Execution

We mitigate the performance overhead of the encoded execution by parallelization. Fig. 3 shows the general approach which is very similar to [11] with two important exceptions:
(1) to reduce the overall overhead and to increase safety, we use static instrumentation instead of dynamic binary instrumentation, and
(2) we introduce speculative variables to decouple the execution of the individual epochs. Speculative variables are discussed in more details in Sec. 3.2.

An encoded execution (see Fig. 3 (b)) introduces, in general, a substantial runtime overhead compared to the unencoded execution (Fig. 3 (a)). To parallelize the encoded execution (Fig. 3 (c)), we execute the original unencoded application within the *predictor* process. The predictor's execution is partitioned into *epochs*. An *executor* process reexecutes a given epoch using its encoded version. Executors run on additional CPU cores in parallel to each other and to the predictor. They synchronize their state using speculative variables to make the approach scalable. The predictor runs up to two orders of magnitude faster than the executors. Hence, it can provide the snapshot for starting the executor of epoch e_{i+1} even if the executor of the previous epoch e_i has not yet finished.

Our parallelization approach is not completely transparent to the application developer. The application developer has to mark potential places in the code where a snapshot could be taken, i.e., a new epoch can be started. Ideally, these snapshot places are executed periodically with constant frequency at runtime.

3.1 Platform Support

At compile time, we first generate two code bases from the unencoded application: the *predictor code base* and the *executor code base*. This stage just duplicates the functions of the original code base and renames them. Second, we instrument both code bases to allow switching from the predictor code base to the executor code base at epoch boundaries. At runtime, the added code rewrites the stack when switching from the predictor's code base to the executor's code base (e.g., it rewrites the return addresses on the stack to point into the executor's code base). After these preparations, the encoding compiler encodes the executor code base. The instrumentation process is the same as for encoding without parallelization.

At runtime, we provide a snapshot mechanism (similar to `fork` [11]) which starts a new executor for each new epoch started by the predictor. The executor replays the same computation for an epoch e as the predictor performed for e but with encoding. Therefore, any input received by the predictor is deterministically replayed in the executors. The input is encoded at runtime by the hand-coded wrappers described in Sec. 2. All externally visible side-effects (issued via system calls) of the predictor are held back until they are verified by the executors. After successfully executing an epoch, an executor explicitly approves it to make its side-effects externally visible. Because the executors are running in parallel, the verification order of system calls might be different from the order in which they were issued in the predictor. We allow out-of-order issuing of system calls by the executors but ensure their in-order retirement.

The deterministic replay and speculative execution of external side-effects is transparent to the encoding compiler and its runtime support. We implemented these two features as kernel-module for Linux similar to Speck [11].

3.2 Speculative Variables

To parallelize the encoded execution of epochs by executors, the executor of epoch e_i starts independently from the state of the executor of its predecessor epoch e_{i-1}. Initially, the executor of e_i contains only the unencoded state from the snapshot of the predictor. Whenever the current state is read in e_i, it is lazily encoded. This approach does not protect against errors in the predictor's execution or the snapshots. Hence, after executing e_i, its executor verifies the *initial encoded* state of e_i against the *final encoded* state of e_{i-1}. By comparing only encoded states, we achieve end-to-end safety.

We implemented the described approach by using *speculative variables*. A speculative variable holds a *value* and optionally an *obligation*. The value is written within the executed epoch, i.e., computed in that epoch. The obligation is the initial value which was read from the snapshot and that needs to be verified at some later point in time. At runtime, the whole encoded state is stored in speculative variables as follows. Every word in memory is assigned to one speculative variable. An epoch starts with an empty set of speculative variables. Speculative variables are created lazily when their corresponding memory addresses are accessed for the first time. When a memory address i is read or written, the value of its speculative variable v_i is read or written, respectively.

A speculative variable v_i is either created by an encoded read from address i or an encoded write to i. When created by a write, the value of v_i is set to the (already) encoded value given to the write. A speculative variable created by a write does not have an obligation. If v_i is created by a read, the unencoded value at address i is read from the predictor's snapshot at the start of the current epoch e_i. This unencoded value is then encoded and written to the value of v_i and to its obligation. Subsequent accesses to v_i do not touch its obligation.

At the end of epoch e_i, all obligations created in the encoded executor of e_i are checked against the final state of the encoded executor of the preceding epoch e_{i-1}. Therefore, the executor of e_{i-1} writes its final encoded state into a global view shared by all executors. At the start of the application, the global view contains the encoded initial state of the application. The executor E_i of e_i waits for the executor of e_{i-1} to terminate. Then E_i verifies all obligations of e_i against the global view. If this verification fails, the application is stopped. In the future, we want to retry the execution of the current epoch to tolerate transient faults. After the verification, E_i updates the global view with the current values of all speculative variables of e_i.

4 Evaluation

We evaluated our approach using four applications: (1) `md5` calculates the md5 hash of a string, (2) `primes` implements the Sieve of Eratosthenes, (3) `tcas` is an open-source implementation of the traffic alert and collision avoidance system [1] which is mandatory for airplanes, and (4) `pid` is a Proportional-Integral-Derivative controller [23].

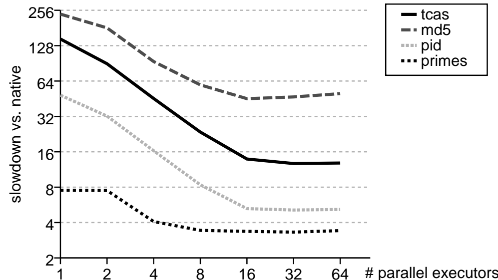

Fig. 4. Slowdowns of sequential and parallelized AN-encoded applications.

Performance. We measured the performance as slowdown of the runtime of the AN-encoded application over the runtime of the unencoded application. Fig. 4 depicts the slowdown of the AN-encoded applications compared to their unencoded versions using different amounts of parallelization by restricting the number of maximally parallel executed AN-encoded executors (x-axis). All tests were executed on 64-bit Fedora 10 running on a 16-core machine.

The sequential slowdowns (1 parallel executor in Fig. 4) range from 7.5 to 238. The more expensive encoded operations are used by an application, the larger the slowdown becomes. Especially `md5` uses many bitwise logical operations and thus experiences a larger slowdown.

Using two parallel executors does not halve the slowdown of the sequential execution because parallelization itself generates overhead by forking new executors, encoded switching from predictor code base to executor code base, and checking the obligations. Starting with 2 parallel executors, the slowdown decreases linearly with the number of parallel executors. Between 8 and 32 parallel executors the decrease stagnates since we are then overloading our 16-core machine. The more overhead the AN-encoded version generates the better does its parallelization scale. With 16 cores the average slowdowns of the tested AN-encoded applications can be reduced from 110 in the sequential case to 16 in the parallelized case. Thus, parallelizing our AN-encoded applications using 16 cores makes them at best 11.5(tcas), worst 2.3(primes) and on average 6.9 times faster.

Error Detection. Fig. 5 shows the results of our error injection experiments. The used error injection tool we implemented using LLVM. At compiliation time, we insert so-called *trigger points* wherever possible. At runtime we decide at each trigger point if it is triggered, i.e., if an error is inserted. If it is not triggered, execution is carried on as in the errror-free case. The caption of Fig. 5 describes the implemented error model.

While unencoded applications show high rates of undetected errors (*incorrect output*), this is not the case for AN-encoded programs. The highest number of *incorrect output* with 7%, we see for md5 with *faulty operations* while the highest number for unencoded programs is 71% for md5 with *exchanged opera-*

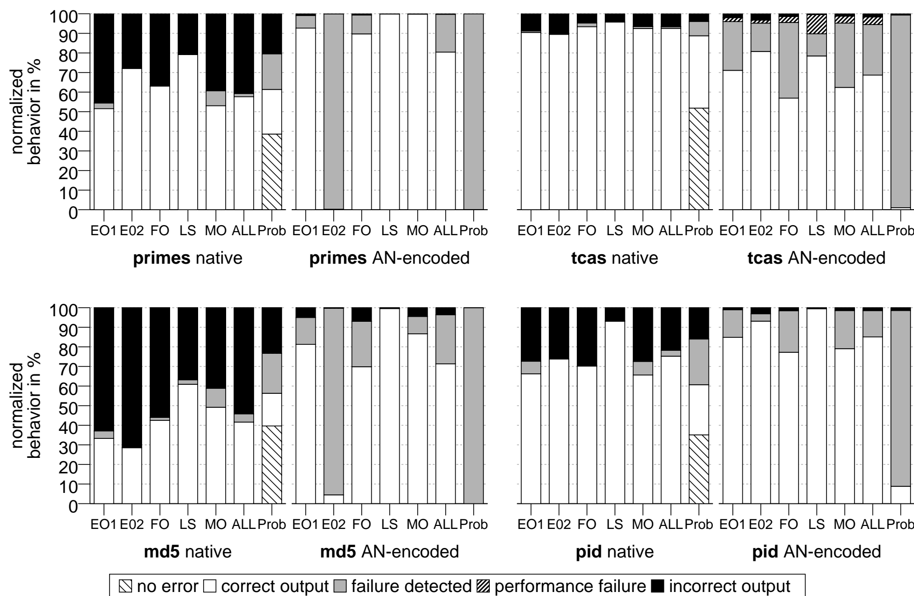

Fig. 5. We inserted the following transient error types: **exchange operands (EO1)**: A different but valid operand is used. **exchange operators (EO2)**: A different operator is used, e.g., a plus instead of a minus. **faulty operations (FO)**: The result of an operation is modified by bit-flips. **lost stores (LS)**: A store operation is omitted. **modify operands (MO)**: An operand is modified by bit-flips. For each example program 10,000 runs with the same input were executed for each error type. In each run exactly one error of that type was inserted. **ALL** represents the average over all those experiments. **Prob**, however, shows the results for probabilistically executed injections of all types. At each possible injection point, we decided if an error should be injected. The same error probability was used for the unencoded and the AN-encoded version. This results in more errors being injected into the AN-encoded version because of its larger code size.

tors. For the probabilistic error injection it even goes down to in average 0.5% of undetected errors compared to 15.9% for the unencoded execution.

In the case of *failure detected*, we either detected an invalid code word and stopped the application, or modified data led to another inconsistency in the program causing a crash. The AN-encoded version always has a higher amount of *failure detected* than the unencoded runs. The probabilistic runs (**Prob**), where multiple errors are injected in nearly all AN-encoded runs, result in a higher percentage of failures detected because the more errors are injected, the higher is the probability of detection [20].

A large part of the injections produced *correct output* which does not differ from the output of the error-free run. For three of the four AN-encoded versions, the amount of such runs increases. Especially with the AN-encoded versions of

the `tcas` benchmark, we see *performance failures*, i.e., the application runs much longer than the error-free run. `tcas` contains several loops whose conditions contain comparisons with 8-bit integers. Encoding increases the size of those values to 64 bit. Injected errors probably increase the contained functional value. This results in longer running loops since AN-encoded comparisons currently do not check the code of their operands. Code checking is expensive and especially for control systems such as `tcas` or `pid` it might be cheaper to use a watchdog to check for liveness.

No error marks those runs of the probabilistic injections into which no error was injected. This is the case for 35% to 52% of the native runs but for none of the AN-encoded runs. Obviously, the increased code path lengths makes it more probable of being hit by an error. But on the other hand AN-encoding makes it possible to prevent erroneous output to a large extent.

5 Related Work

Error Detection For an overview of hardware approaches for hardware error detection see our previous work [14]. They all have in common that custom hardware is typically very expensive and not often adapted to new faster technologies. Our intention is to provide a software-implemented error detection mechanism which provides detection guarantees independent of the used hardware. This allows to use up-to-date hardware in safety-critical systems which require certification.

Control flow checking approaches such as [4, 16] in contrast to AN-encoding can detect invalid control flow for the executed program, that is, execution of sequences of instructions which are not permitted for the executed binary. Modified data values, that an AN-code detects, will remain undetected.

In [15, 22] invariants contained in the executed program are used to check the validity of the generated results. Thereby good error detection can be provided but for most programs it is difficult—if not impossible—to design invariants with good failure detection capabilities and to assess the quality of these invariants.

Replicated execution and comparison of the obtained results as for example used by [18, 3] provide no guarantees with respect to permanent hardware errors or soft errors which disturb the voting mechanism.

ED4I [12] uses also an AN-code but the authors choose a factor A which is a power of two whenever a program contains logical operations. Thereby, logical operations become easily encodable but the detection capabilities are reduced immensely because the resulting code cannot detect bit flips in the higher order bits of data values. But those contain the original functional value. [6] applies the AN-code only to registers and not to memory and only to operations which easily can handle encoded values such as additions and subtractions. In the end that leaves supposedly only small parts of applications which are AN-encoded. As should be expected their fault injection experiments show a non-negligible amount of undetected failures for most of the tested applications. Both [12] and

[6] do not discuss overflow problems with AN-codes which we pointed out and solved in [19].

Parallelized Checks Recent work [11, 13, 17] exploits modern multi-core systems for reducing the overhead of expensive runtime checks. As in our parallelization framework the execution is split into epochs. The application runs as speculator on a single core. Whereas, the other cores replay the execution of the speculator using multiple parallel checkers. But there are also major differences. Speck [11] and SuperPin [17] rely on dynamic binary instrumentation. Instrumenting AN-encoding at runtime is less safe as a static instrumentation [21]. While Speck changes the whole OS kernel, we implemented a kernel module which is much easier to deploy and maintain. SuperPin does not have a speculation support on the syscall level. Thus, erroneous output cannot always be blocked *before* it is verified.

Different AN-encoded checker epochs have to share state because it is required that predecessors verify the error-freeness of the start state of their successors. Our understanding is that SuperPin does not support sharing state between parallel running checker epochs. Speck and parallel DIFT [13] merge the checking data gathered by the checkers in a separate thread into sequential order. This thread can become a bottleneck when the number of checkers increases. Parallel DIFT uses a hardware extension [7] to stream data between cores. We do not rely on any specialized hardware but implemented state sharing using *speculative variables.*

6 Conclusion

We demonstrated that AN-codes can be used to reduce the rate of undetected incorrect output for frequently occurring error events from 15.9% to 0.5% and for rare error events with only one injected error per run from 30.8% down to 1.7%. The remaining undetected errors can be tackled by using the more powerful but also more expensive AN-code with signatures as introduced by Forin in [9].

References

1. The Paparazzi Project. http://paparazzi.enac.fr/wiki/Main_Page.
2. A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. In *Transactions on Computers*, 1971.
3. C. Bolchini, A. Miele, M. Rebaudengo, F. Salice, D. Sciuto, L. Sterpone, and M. Violante. Software and hardware techniques for SEU detection in IP processors. *J. Electron. Test.*, 24(1-3):35–44, 2008.
4. Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.
5. Shekhar Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 2005.

6. Jonathan Chang, George A. Reis, and David I. August. Automatic instruction-level software-only recovery. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*. IEEE Computer Society, 2006.
7. Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*. IEEE Computer Society, 2008.
8. Anand Dixit, Raymond Heald, and Alan Wood. Trends from ten years of soft error experimentation. In *System Effects of Logic Soft Errors (SELSE)*, 2009.
9. P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, pages 79–84, Sept 1989.
10. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization (CGO)*. IEEE Computer Society, 2004.
11. Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. *SIGARCH Comput. Archit. News*, 2008.
12. Nahmsuk Oh, Subhasish Mitra, and Edward J. McCluskey. ED4I: Error detection by diverse data and duplicated instructions. *IEEE Trans. Comput.*, 51, 2002.
13. Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, USA, 2008. ACM.
14. Ute Schiffel, Martin Süßkraut, and Christof Fetzer. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *The 28th International Conference on Computer Safety, Reliability and Security (SafeComp 2009)*, 2009.
15. V.K. Stefanidis and K.G. Margaritis. Algorithm based fault tolerance : Review and experimental study. In *International Conference of Numerical Analysis and Applied Mathematics*, 2004.
16. Ramtilak Vemu and Jacob A. Abraham. CEDA: Control-flow error detection through assertions. In *IOLTS '06: Proceedings of the 12th IEEE International Symposium on On-Line Testing*. IEEE Computer Society, 2006.
17. Steven Wallace and Kim Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *5th Annual International Symposium on Code Generation and Optimization*, pages 209–217, San Jose, CA, March 2007.
18. Cheng Wang, Ho seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *International Symposium on Code Generation and Optimization (CGO)*, 2007.
19. Ute Wappler and Christof Fetzer. Hardware failure virtualization via software encoded processing. In *5th IEEE International Conference on Industrial Informatics (INDIN 2007)*, 2007.
20. Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *The 26th International Conference on Computer Safety, Reliability and Security (SafeComp 2007)*, 2007.
21. Ute Wappler and Martin Müller. Software protection mechanisms for dependable systems. *Design, Automation and Test in Europe (DATE '08)*, 2008.
22. Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *J. ACM*, 1997.
23. Tim Wescott. PID without a PhD. *Embedded Systems Programming*, 13(11), 2000.