Silent Data Corruption Detection with Program Synthesis

Kerry Tu*
Weixin Yu*
kjwtu@stanford.edu
weixinyu@stanford.edu
Stanford University
Stanford, California, USA

Abstract

Reliability issues in hardware circuits have become increasingly important in the age of data and computation explosion. Hyperscalers are reporting Silent Data Corruptions (SDCs), computation errors where the only symptom is an incorrect result. Those errors are extremely difficult to locate and debug. Existing software-level proactive testing approaches either take a long execution time or focus on only part of the sources of hardware defects.

We present our automated tool, packaged as an LLVM compiler pass, that generates alternative but functionally equivalent instruction sequences using Program Synthesis techniques. The compiler pass transforms a valid LLVM program into a functional consistency check that enables SDC detection. The tool is able to localize the specific LLVM instruction where silent data corruption occurs.

We also implemented an error injection and detection framework at the LLVM-IR level. The errors are inserted on top of the 6 instrumented tests generated from C programs at the LLVM-IR level. Our results demonstrate the effectiveness of our approach in detecting SDCs with acceptable performance overhead (within 3%). The instrumentation time for a program with around 700 lines of code is around 1 minute.

CCS Concepts: • **Computer systems organization** → **Reliability**; **Redundancy**; • **Computing methodologies** → *Equation and inequality solving algorithms.*

*Both authors contributed equally to this research.

CS357S '25, Stanford, CA

ACM ISBN 357-3-5735-7357-3/2025/03
https://doi.org/3573573.5735735

Keywords: Silent Data Corruptions, Program Synthesis, Rosette, Functional Consistency, LLVM, Error Injection, Instruction Duplication

ACM Reference Format:
Kerry Tu and Weixin Yu. 2025. Silent Data Corruption Detection with Program Synthesis. In *Proceedings of CS357S Conference (CS357S '25).* ACM, New York, NY, USA, 7 pages. https://doi.org/3573573.5735735

1 Introduction

Silent Data Corruptions (SDCs) are computation errors in which the only symptom is an incorrect result. The error can only be detected after accessing the database containing that row. Therefore, it is very difficult to know when, where, or how the corruption occurs.

Current SDC mitigation solutions by both datacenter operators and researchers employ the common idea of proactive testing. Google's SiliFuzz [9] repeatedly tests cores using randomly generated tests. To reduce the test size, Vega [7] leverages knowledge from the hardware and generates targeted tests focused on transistor aging. ITHICA [6] employs an intra-thread instruction checking technique by inserting replicated instructions and performing consistency checks.

However, it is likely that duplicated instructions will undergo the same control path and be executed by the same functional unit in the underlying hardware architecture. If this happens to be true, the result of a duplicate instruction will always be equal to the original, defeating the purpose of being a consistency checker.

We inherit the intra-thread instruction checking from ITHICA [6] but take a different approach. Instead of duplicating each instruction, we generate alternative instruction sequences that perform the same architectural function from the original instruction using program synthesis. As a result, our approach works even if defect-induced SDCs manifest consistently.

In particular, we generate replacement code snippets at the LLVM-IR level using Rosette. Our framework performs program synthesis and instruments the code at LLVM-IR level and is thus directly applicable to most architectures. Moreover, our implementation enables users to tune the

number of alternative snippets and the maximum length of generated alternative instruction snippets.

In addition, we implemented an error injection and detection framework at the LLVM level to evaluate the effectiveness of our tool. On each execution of the modified program, a bit manipulation is introduced at a tunable probability to emulate SDC-manifesting hardware. The framework is able to pinpoint the line number of the original instruction.

To evaluate the effectiveness of alternative code snippet insertion and functional consistency check on early detection of SDCs, we run our framework on a test suite consisting of six C programs that implement common algorithms and data structures. We show that the methodology can generate programs capable of detecting SDCs with low overhead and generation time.

Overall, we make the following contributions:

- We present our framework and tool that generates and inserts alternative but functional equivalent code snippets back into original code at the LLVM-IR level. We implement the tool as an LLVM compiler pass.
- We implemented an error injection and evaluation framework at the LLVM-IR level to model the behavior of defected hardware. The evaluation framework is able to pinpoint the line of code where the corruption occurred, resolving the inherent difficulty of locating errors for SDCs.
- We demonstrate the effectiveness of our approach with 6 C programs. We show that the overhead of both program synthesis and execution is acceptable.

2 Background

2.1 Silent Data Corruption

Silent data corruptions (SDCs) are undetected errors generated in computing, caused by hardware defects which are not consistent over inputs, program execution, or the processor's life cycle. Design bugs are not included in the category. Therefore, targeted mitigation techniques other than traditional silicon validation approaches like Quick Error Detection (QED) are necessary.

The only symptom of SDCs is incorrect data. Therefore, errors can propagate across the stack silently. Only after an unpredictable amount of time when the data are accessed again can the error be detected. This has made SDCs very hard and laborious to trace.

SDCs have had a negative impact on large-scale infrastructure services. For example, Meta reported errors that have caused data loss and cost months of debug engineering time [2].

The inherent cause of SDCs is the mismatching assumption made by software applications that processors always behave as expected with respect to their architectural specifications. While there is no existing technique to eliminate all hardware manufacturing defects yet, researchers aim to address this critical issue by proactively testing the underlying hardware at the software level.

2.2 Related Work

Previous research by Dixit [2] has surveyed various methods to debug SDCs, in both the hardware and software spaces. Some recommended hardware mitigations include the use of protected datapaths, specialized screening, understanding at-scale behavior, and architectural priority. The paper also offered several options for software mitigations for SDCs, including redundancy and fault-tolerant libraries. In Meta's own research [1], they created two suites for testing hyperscalar systems: Fleetscanner and Ripple. Fleetscanner focused on out-of-production testing, in which the servers are in maintenance and not currently running productive workloads, while Ripple focused on in-production testing, which allowed testing without halting running systems. Furthermore, Fiala's RedMPI library [3] has been shown to be effective in detecting and correcting soft errors in MPI message passing applications, using a mix of redundancy and consistency protocols. Hari's work [5] attempts to improve the efficiency of SDC detection systems by analyzing what types of code sections caused the most SDCs (90% of SDCs were caused by 5.4% of static instructions). With this analysis, they successfully improved efficiency by placing detectors at these fault-prone code sections (comparing similar computations, checking value equality, performing range checks, and performing mathematical tests), compared to a purely redundancy-based system.

3 Approach

We design, implement, and evaluate a framework that helps detect silent data corruptions (SDCs). Figure 1 provides the design overview for the framework. It takes in a valid program in LLVM-IR, generates functionally equivalent instructions for each arithmetic operations, instruments the program, and executes repeatedly on a server to detect SDCs.

3.1 Alternative Instruction Generation

We perform the program synthesis at per-instruction granularity, while allowing the generated instruction sequence to contain multiple instructions. In other words, there are at most two source registers and one destination register in both the original and modified programs. For each single original instruction, Rosette program synthesis gives a recursive expression representing the generated sequence. Another translation we did was to translate the expression into a list of LLVM-IR instruction sequence.

3.2 Program Instrumentation

To achieve proper silent data corruption detection, the code created using program synthesis must be inserted back into

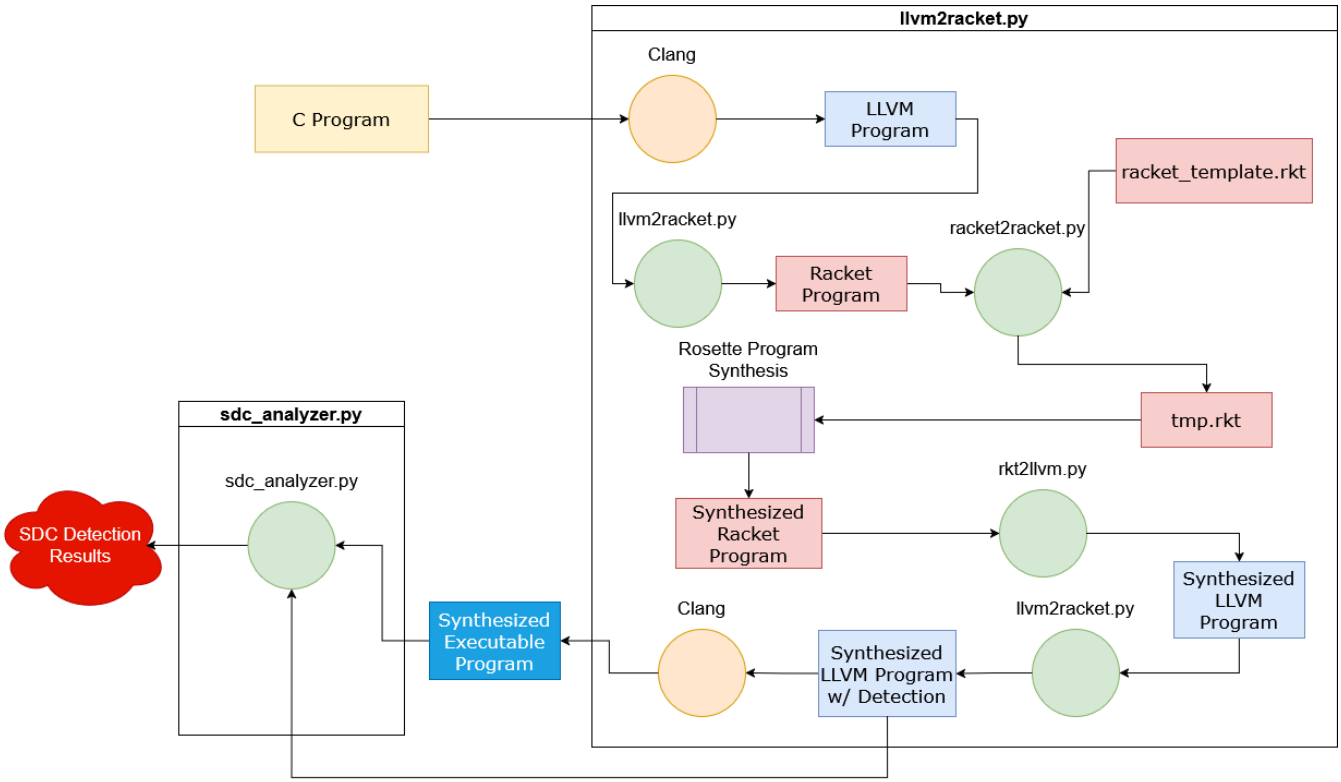

Figure 1. Toolflow for SDC Detection with Program Synthesis

the original Clang-generated LLVM code, and the result of the synthesized code must be equal to the result of the original line. If they match, then there is no detected silent data corruption. However, if they differ, the program detects an SDC and must halt execution. Ideally, the specific line from which the SDC originated should be able to be pinpointed, so that localized debugging or analysis may be performed if desired, post-SDC detection. Some language conversions are necessary throughout the process, as the input is a C program and the output is modified LLVM code or executable. The Rosette program synthesis implementation requires input in the Racket language and also outputs synthesized code in Racket.

4 Implementation

4.1 Alternative Instruction Generation

In our implementation, we used Rosette, a solver-aided programming language that extends Racket, for program synthesis.

To synthesize instructions, Rosette needs both a grammar template of the instruction we are generating and a specification. The grammar template is a recursive definition in approximate Backus-Naur form (BNF), and the specification is defined as the semantics of the original instruction.

As both the grammar template and the specification depend on the specific LLVM-IR instructions, we decided to generate the racket program dynamically using Jinja2, a templating engine. We decide not to create a lookup table from general LLVM instructions to alternative instructions in order to accommodate diverse translations stemming from constant parameters within instructions. For example, there is no solution for a general `mul i32 %reg1, %reg2`, but various solutions exist for `mul i32 %reg1, 2`, which is equivalent to `add i32 %reg1, %reg1` as well as `shl i32 %reg1, 1`.

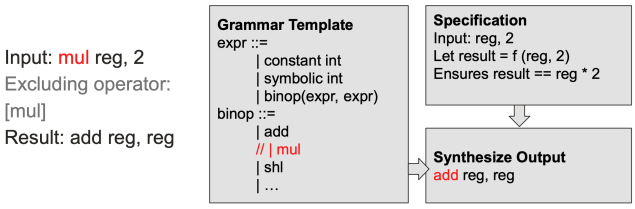

Figure 2. A simple program synthesis example.

To avoid trivial synthesis result such as a simply duplicated instruction, alternative instructions are constrained to use sets of LLVM-IR binary operators disjoint with the original instruction. As shown in Figure 2, for the instruction `mul i32 %reg, 2`, the `bvmul` operator is excluded from the

grammar template to avoid generating duplicate instruction. The specification is defined as multiplying by 2, and the result is `add i32 %reg, %reg`.

The alternative code snippet generated by Rosette can contain multiple instructions, depending on the depth set with the grammar template. We thus implemented a compiler pass using recursive functions that translates nested Racket expressions into a list of LLVM-IR instructions.

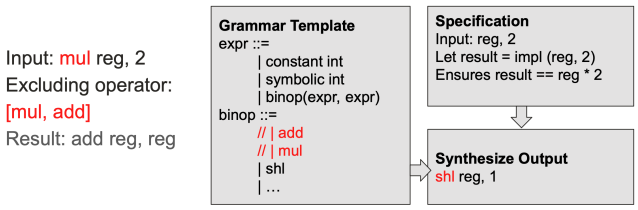

Figure 3. Program synthesis for generating more alternative instruction snippets.

It is reasonable to ask for as many different alternative instructions as possible, in order to exploit different functional units. We partially tackle the problem by exploiting our dynamic code generation approach. Between Rosette program executions on the same LLVM instruction, we modify the grammar template for the generated program according to the result from each iteration. We maintain an *exclusion list* in Python, keeping track of all operators that have already been used in either the original instruction or the previously generated alternative instructions. The set of arithmetic operators listed in the Rosette grammar template for the next iteration is then calculated as the difference between the set of all arithmetic operators and the *exclusion list*.

Following the example in Figure 2, as shown in Figure 3, we add the operator used in the generated solution in Figure 2 to the exclusion list and exclude it from the grammar template while keeping the specification unchanged. The synthesized output on this run becomes `shl i32 %reg, 1`, yet another alternative instruction.

4.2 Program Instrumentation

The toolflow of the SDC detection is shown in Figure 1, and is completely automated. The part on the right takes in a C program and generates the instrumented program with the SDC detection mechanism. The part on the left represents the error injection and execution framework, which will be detailed in the Evaluation Methodology.

We begin llvm2racket.py with a C program as input, and this will be the code that the user wants to evaluate or run on servers. Using Clang, the C program is compiled into an LLVM program, which is then converted with the script to a Racket program. For each arithmetic instruction in the program, it converts the instructions from LLVM to Racket using a mapping of instructions, along with the necessary syntax. The single-line expression in Racket is then fed into the Program Synthesis framework discussed in Section 4.1, where an (optional) alternative list of LLVM instructions is returned.

The synthesized list of LLVM instructions (or the replicated original instruction in case no alternatives are found) is then inserted before its corresponding original LLVM instruction. Additional comparison code is also appended after that compares the original LLVM instruction's result with the result of the synthesized code by calling an `sdc_check` function. This function exits the program with a particular error code corresponding to the error line if the input registers differ.

Once this is done with all arithmetic instructions in the original code, the complete synthesized LLVM program with SDC detection is compiled into an executable using Clang.

Listing 1. Simple C Test Program (pow2).

```c
#include <stdio.h>
int main() {
    int p = 4;
    int result = 1;
    for (int i = 0; i < p; i++) {
        result = result << 1;
    }
    printf("Result: %d\n", result);
    return 0;
}
```

To illustrate the toolflow more clearly, we show the toolflow outputs from a simple C test program above. This program, pow2, calculates and prints the value of 2 to the power of p, where p is 4.

As expected, the script detects two arithmetic operations, `%11 = shl i32 %10, 1` and `%14 = add nsw i32 %13, 1`. They are converted to Racket codes `%dest0 = (bvshl %10 (int 1))` and `%dest1 = (bvadd %13 (int 1))`, respectively. These are then used to populate the Racket template to generate alternative Racket instructions that are functionally consistent. We can reason that the shift left by 1 is functionally equivalent to adding the input twice, and subtracting -1 from a register and then shift right by 0 is equivalent to the original add 1.

The script appends the synthesized code, along with its corresponding `sdc_check` function, to a modified LLVM files and then compiles with Clang to executables. Figure 4 better illustrates this insertion. At the end of the script, the executable is run once. If there are no errors, it will print "Execution Successful" with a return code of 0.

5 Evaluation Methodology

To evaluate the effectiveness of our methodology, we artificially introduce errors into the program during runtime at a set probability. The error injection and execution framework take the synthesized LLVM and outputs a set of SDC detection results, including the number of SDCs detected

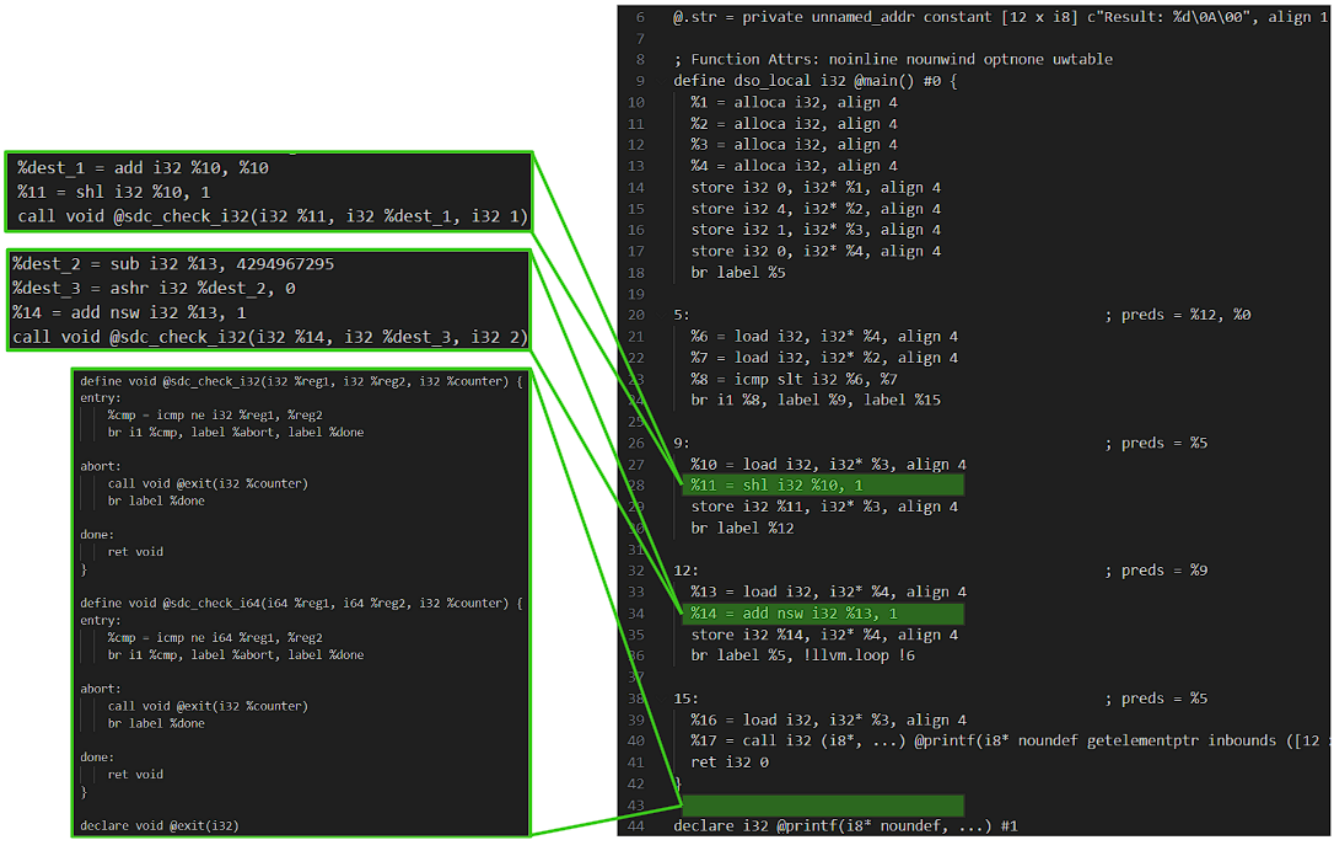

Figure 4. Insertion of LLVM Modifications (left) into Clang-generated LLVM (right).

and a bar graph displaying the number of errors found on each line. Setting the variable `INTRODUCE_ERRORS` to `False` in `sdc_analyzer.py` disables error injection, which is reserved for running on real servers. With `INTRODUCE_ERRORS` set, the script will parse the synthesized LLVM file and generate separate LLVM files and executables, each corresponding to an error injected in one of the program lines. This is done by adding the value 1 to the original result and adjusting the other register names accordingly in the file. We now have a suite of LLVM files and executables to run for each program line error.

We then use the user-set individual line error probability, `ERR_PROB`, to determine the total probability that an error occurs during runtime according to the equation below, where N is the number of possible lines:

$$total\_error\_prob = 1 - (1 - ERR\_PROB)^N$$

The `sdc_analyzer.py` script then runs 10,000 iterations of the executable, and each iteration will have a total_error_prob probability of executing one of the error-injected files, and a (1 - total_error_prob) probability of executing the synthesized executable. Each error-injected file has an equal probability of being chosen. The errors injected are recorded for later evaluation and analysis. Each executable will either run until completion, or until one of the consistency checks fails and returns a non-zero error code. The error code is used to pinpoint the exact line where the SDC occurred.

6 Results

We evaluate our tool by running the packaged LLVM compiler pass on 6 test C programs compiled to LLVM using clang. The error injection framework acts as an additional LLVM compiler pass that generates faulty LLVM code at all arithmetic instructions. The execution framework then executes the modified programs repeatedly, each time with a certain probability of executing each error-injected version or the non-injected version.

The sdc_analyzer.py script generates two temporary modified LLVM files and executables, each corresponding to an error at the specified lines 29 and 38. The total error probability was calculated from the equation in the previous section to be 0.001999 from the individual ERR_PROB of 0.001 and N=2. A total of 222 errors were injected. The number of SDCs found by our SDC detection system was also 222 errors, with the same distribution of errors in the pinpointed lines from the returned error codes, as the detector did not miss any SDCs in this example. Figure 5 displays a graph for the distribution of detected errors.

Table 1. Evaluation result of the SDC detection tool as a compiler pass, running on 6 C programs.

Test Program	#lines in original	#lines in modified	Alternatives generated	Generation Time (s)	Error Detection Rate (%)	Execution overhead (%)
backtrack.c	277	340	6	8.42	100	-0.93
basic_malloc.c	498	569	11	31.19	47	-0.92
bfs.c	724	825	21	60.53	84	0.93
pow2.c	62	110	2	2.74	100	2.73
prime.c	132	180	1	15.21	100	0
priority_queue.c	608	681	12	40.47	55	1.75

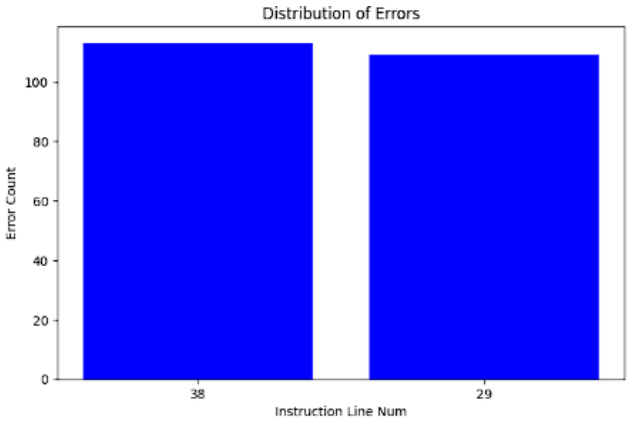

Figure 5. Graph Output from sdc_analyzer.py for pow2.

The program size, number of alternative instruction sequences generated, generation time, execution overhead, and error detection rate for each of the 6 test programs are shown in Table 1. Alternatives generated counts the number of alternatives found through program synthesis. Instructions for which no alternatives are found and simply duplicated in the program are not included. Generation time indicates the total time spent on program synthesis, running locally on laptops. Execution overhead is measured by the ratio of execution time running the modified (but not error-injected) versions of the programs to the original versions 10,000 times.

As shown in Table 1, the generation time is at most around one minute for a program with around 700 lines of code and 21 alternatives generated. The execution overhead is at most 3% in the worst case for all programs tested.

The error detection rate refers to the number of detected corruptions divided by the number of injected errors. It is not 100% because some errors are injected in some control paths of the program that is never executed in the current iteration.

7 Discussion and Limitations

While the instruction insertion tool is presented as a complete LLVM package, both the tool itself and the evaluation methodology have potential for future improvement.

The implementation of our tool inserts alternatives only for arithmetic instructions. Other important units, including memory and conditional instructions, are not checked. The program synthesis module can be extended to model memory and control path operations. For example, a load-word can be translated into two load-half-word instructions. Stores can be translated similarly and then checked with additional round-trip consistency comparison, as proposed in ITHICA [6].

As mentioned in the evaluation section, another limitation is that the total number of introduced errors included errors introduced in control paths that were never executed during the current iteration of the program. This appeared in the results as though the SDC analyzer "missed" the error, but any analyzer would not be able to detect an error not encountered. For more accurate results, we should only count the errors that were encountered during runtime rather than all the errors introduced. A better approximation is to use LLFI [8] as the error injection framework. We did not use LLFI because the repository is old and many packages are deprecated. However, the errors artificially inserted at the LLVM level may not mirror the actual hardware defects. It is more accurate to use the real (defected and non-defected) servers and observe if corruptions are detected accurately.

The test programs used in the evaluation part are simple C programs that are not comparable in computation intensity to programs running on hyperscaler's server fleets. Complex tests, such as cpu-check [4], can be added as new test suites that target SDCs in the real world.

Acknowledgments

To Caroline Trippel and Rachel Cleaveland for their help throughout the class CS357S Formal Methods for Computer Systems, Winter 2025, at Stanford University.

References

[1] Harish Dattatraya Dixit, Laura Boyle, Gautham Vunnam, Sneha Pendharkar, Matt Beadon, and Sriram Sankar. 2022. Detecting silent data corruptions in the wild. arXiv:2203.08989 [cs.AR] https://arxiv.org/abs/2203.08989

[2] Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245* (2021).

[3] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1109/SC.2012.49

[4] Google. n.d.. CPU-check torture test. https://github.com/google/cpu-check. Accessed: 2025-03-17.

[5] Siva Kumar Sastry Hari, Sarita V. Adve, and Helia Naeimi. 2012. Low-cost program-level detectors for reducing silent data corruptions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 1–12. doi:10.1109/DSN.2012.6263960

[6] Caroline Trippel Ioanna Vavelidou (and maybe others). 2025. ITHICA: Intra-Thread Instruction Checking Approach for Defect-Induced Silent Data Corruptions. In *N/A*.

[7] Madeline Burbage Theo Gregersen Rachel McAmis Freddy Gabbay Baris Kasikci Jiacheng Ma, Majd Ganaiem. 2024. Proactive Runtime Detection of Aging-Related Silent Data Corruptions: A Bottom-Up Approach. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[8] Qining Lu, Mostafa Farahani, Jiesheng Wei, Anna Thomas, and Karthik Pattabiraman. 2015. Llfi: An intermediate code-level fault injection tool for hardware faults. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 11–16.

[9] Kostya Serebryany, Maxim Lifantsev, Konstantin Shtoyk, Doug Kwan, and Peter Hochschild. 2021. Silifuzz: Fuzzing cpus by proxy. *arXiv preprint arXiv:2110.11519* (2021).

Received 18 March 2025