RELYZER+: AN OPEN SOURCE TOOL FOR APPLICATION-LEVEL SOFT ERROR RESILIENCY ANALYSIS

BY

KHALIQUE AHMED

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Sarita V. Adve

ABSTRACT

In the modern era of computing, processors are increasingly susceptible to soft errors. Current solutions in both hardware and software enable error detection and correction. Some of these errors, however, go unnoticed by detectors and manifest as silent data corruptions (SDCs) at the application level. Injecting errors into the system and evaluating the outcomes is one method to uncover SDC-causing errors and determine an application's overall resilience to soft errors. The number of possible locations that errors may appear in is large, therefore requiring many injection experiments.

One resiliency analysis tool, Relyzer, addresses this issue by performing a comprehensive program analysis to create a small subset of the error injection experiments that can account for the entire application. The limitation of Relyzer is that current analysis can only be performed on one hardwarware instruction set architecture (ISA). Software is usually compiled to multiple ISAs in order to support users with varying hardware configurations.

The primary contribution of this thesis is building *Relyzer+*, an open source version of Relyzer implemented using the gem5 simulator. This enables the capability to analyze multiple ISAs and consequently support multiple hardware configurations in the long-term. Specifically, in this work, we develop support for x86. We also evaluate applications across ISAs by generating error resiliency profiles for both x86 and SPARC. After studying five workloads from different domains, we find that in general, application soft error resiliency varies based on the selection of the ISA. The percentage of static instructions that yield SDCs is, on average, 68% for x86 and 60% for SPARC, for the applications we studied. Furthermore, this work opens doors to future research in application-level soft error resiliency analysis.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grant CCF-1320941, the Center for Applications Driving Architectures (ADA), one of six centers of JUMP, a Semiconductor Research Corporation program co-sponsored by DARPA, and by the Center for Future Architectures Research (C-FAR), one of the six centers of STARnet, a Semiconductor Research Corporation program sponsored by Marco and DARPA.

I would like to thank my advisor, Professor Sarita Adve for her continued support throughout these past few years. With her guidance, I overcame hurdles in my research and progressed to the point where I am today. I would also like to thank my collaborators, Professors Chris Fletcher, Darko Marinov, and Sasa Misailovic for providing me with additional insights to strengthen my work. I owe a lot to my other collaborators, Abdulrahman Mahmoud and Radha Venkatagiri, who were two members from my research group that always came to my aid at a moment's notice.

I would like to thank John Alsop, Lin Cheng, Adel Ejjeh, Muhammad Huzaifa, Weon Taek Na, Gio Salvador, and Matt Sinclair, who were other members from the group that I learned from. Even Siva Hari, who graduated before I joined the group, helped me along the way, and he deserves credit for establishing the foundation for this work.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

As the transistor count and complexity of a processor increases, the likelihood of soft errors caused by high-energy alpha particle strikes also increases [1, 2]. This is problematic because such trends in transistor scaling encompass a wide array of computing platforms. Therefore, developing reliable systems no longer pertains to niche workloads alone. Moreover, in the majority of scenarios, execution is error-free, so overprotecting an application may result in significant overhead. Both of these problems necessitate development of reliable systems that incur minimal overhead.

Hardware and software engineers currently combat soft errors to a certain extent by incorporating error correction and detection mechanisms into the target system [3, 4, 5, 6]. Despite efforts to develop and deploy with soft error resilience in mind, the nature of the applications themselves may dictate how errors manifest and propagate throughout the system [7, 8, 9, 10].

Emerging workloads such as recognition, mining, synthesis (RMS) and several machine learning algorithms are resilient by nature, as they tolerate imprecision and accuracy in their computation by design [11, 12]. On the contrary, software used in safety-critical systems such as avionics and autonomous vehicles may be more sensitive to soft errors, especially *Silent Data Corruptions* (SDCs). An SDC is a classification of errors that are undetectable during execution and corrupt program output. Therefore, SDCs are an important subset of error outcomes that need to be addressed when analyzing a program's error resilience.

A common approach for uncovering SDCs is to simulate application execution, introduce errors by corrupting state during runtime, and empirically evaluate the outcome. Within the simulation, we define an error site as any potential location in which an error may be injected. Applications can execute trillions of instructions, and injecting into every error site is computationally expensive. Therefore, an effort must be made to select appropriate error sites to inject into that provide enough insight on the portions of the application that are most vulnerable to SDC-causing soft errors.

One technique to error site selection involves taking a statistical approach and sampling error sites [13, 14]. The main drawback to this method is that sampling such error sites does not provide any insights for SDCs outside the sampled regions. Consequently, the statistical approach lacks comprehensive analysis of an application from end-to-end.

An alternative is to employ heuristics that leverage knowledge of the execution behavior such as control flow or load/store patterns to categorize a multitude of error sites into one outcome category. There are several modern fault injection tools that take these heuristics

into account to group error sites together and perform error injection in only one representative from within that group [15, 16, 17, 18, 19, 20]. The remainder of error sites are pruned, or equalized to the injection outcome of the representative.

One specific tool, Relyzer [16], provides a detailed and comprehensive outlook of application resiliency. Relyzer orchestrates control and store-based heuristics in addition to precise techniques to prune a significant portion of the error sites. Analysis is done at the instruction set architecture (ISA) level. This is advantageous because users can abstract away most of the processor's low-level hardware execution. The ISA-level analysis provides insights into outcomes from errors that manifest as single bit-flips in a processor's user-visible architectural registers. Subsequently, by modeling errors in this fashion, performing bit-flips in these registers indicates the assembly instructions that are vulnerable to SDCs.

Although Relyzer is robust, one of it main limitations is its infrastructure. The tool relies on Wind River Simics [21], a proprietary full system simulator. The infrastructure is also designed to handle only applications compiled for the SPARC ISA. The restrictions imposed from both the simulator and ISA make adoption of the tool challenging.

The main contribution of this work is the development of *Relyzer+*, an expansion to Relyzer that enables support for more ISAs, beginning with x86. To perform the aforementioned experimental infrastructure reconstruction, we build our foundations on the open source gem5 simulator [22]. This facilitates the inclusion of future ISAs into the tool because the computer architecture research community uses gem5 as one of its main platforms for testing new hardware designs.

Building Relyzer+ requires significant engineering effort in order to support x86 analysis on gem5. Relyzer's original algorithms assume constant register size and instruction encoding length, which is not the case with x86. We also require the development of an injector module within gem5 that performs single bit-flips in the CPU registers at the appropriate instance in time. Even with the added complexity, we succeed in implementing Relyzer+ on both a different simulation platform and ISA.

We then perform fault injection experiments on both x86 and SPARC. For preliminary evaluation, we analyze five workloads from different domains. Each application undergoes Relyzer+'s pruning techniques to reduce the number of error sites. We perform an additional validation of the heuristics to verify that the grouping of error sites yield the same outcome. After performing injections in the remainder of error sites, we analyze the distribution of outcomes over all static instructions and compare the percentage of SDC-causing instructions between the two ISAs.

The average accuracy, or correct outcome prediction rate for each heuristic on x86 is above 96%, indicating strong validation. From prior experiments [20], SPARC also exhibits

a high validation accuracy. Additionally, we calculate the pruning effectiveness for each application, which indicates the fraction of error sites pruned out of the total error sites. Across all studied workloads, the average pruning effectiveness is 95% for x86 and 83% for SPARC. After performing injection experiments, on average, Relyzer+ identifies that approximately 68% of static instructions yield SDCs for x86, and 60% of static instructions yield SDCs for SPARC.

The remainder of this thesis is organized as follows: First, we elaborate on concepts related to ISAs, Relyzer's pruning techniques, and the gem5 infrastructure. Next, we describe Relyzer+'s implementation of program analysis and error injection using gem5. The subsequent chapter describes the methodology of validating heuristics and evaluating workloads using the new infrastructure. We then analyze the preliminary results of our error injection experiments. In the remaining chapters, we discuss the relationship between Relyzer+ and existing tools and conclude by suggesting future research directions.

CHAPTER 2: BACKGROUND

In this section, we introduce the key concepts necessary to grasp the contributions established in this work.

2.1 INSTRUCTION SET ARCHITECTURES

Assembly is a low-level programming language that incorporates instructions that closely reflect the machine code executed by a CPU. The actual instruction set architecture (ISA) enumerates all possible assembly instructions that a programmer may require.

The two primary categories of ISAs are elaborated below:

1. **Reduced Instruction Set Computing (RISC)** - Each instruction's encoding is fixed in length. Operations are relatively simplistic because they execute in one CPU clock cycle. SPARC is an example of a RISC ISA.

   Figure 2.1 illustrates instructions from a disassembled SPARC binary. The leftmost column gives the PC associated with a particular instruction. The information in the middle, namely the set of 4 bytes followed by the translated operation (or opcode), provides all of the required encoding for a CPU to process and execute an instruction. Register operands (denoted by a "%" symbol) and constant data values appear in the rightmost column. Each PC offset is 4 bytes because of the fixed instruction length.

   The advantage to using a RISC ISA is that the other aspects of the CPU become simpler to design, especially the decoding unit that is responsible for translating each instruction and issuing the necessary commands for correct execution.

2. **Complex Instruction Set Computing (CISC)** - Each instruction can vary in length and subsequently can perform multiple operations before incrementing the PC. x86 is an example of a CISC ISA.

   Figure 2.2 depicts sample instructions for x86. The key difference between RISC and CISC becomes clear when examining the second column, signifying different lengths of instruction encoding.

   A benefit to using CISC is that few instructions can essentially decode to powerful sequences of computation. Another effect as evidenced in Figure 2.2 is that code becomes more compact since fewer bytes may be required for simpler operations such as pushing a register onto the stack.

```
main()
    0x100001b30:        9d e3 bd 30  save      %sp, -0x2d0, %sp
    0x100001b34:        90 07 a7 ef  add       %fp, 0x7ef, %o0
    0x100001b38:        27 00 04 00  sethi     %hi(0x100000), %l3
    0x100001b3c:        40 04 13 a9  call      +0x104ea4     <0x1001069e0>
    0x100001b40:        92 10 20 00  clr       %o1
    0x100001b44:        f8 5f a7 ef  ldx       [%fp + 0x7ef], %i4
    0x100001b48:        91 3e 20 00  sra       %i0, 0x0, %o0
    0x100001b4c:        a4 14 e0 05  or        %l3, 0x5, %l2
    0x100001b50:        a3 2c b0 0c  sllx      %l2, 0xc, %l1
    0x100001b54:        21 00 04 00  sethi     %hi(0x100000), %l0
```

Figure 2.1: Example instructions of the SPARC ISA.

The largest drawback to CISC is its complexity. x86 in particular has small registers that access part of a larger register. For example, *%eax* refers to the lower 32 bits of *%rax*. The variable-length instruction encoding is associated with a complex decoder unit that requires multiple cycles to decode each instruction.

2.2 RELYZER

Relyzer is a tool that incorporates program analysis and heuristics to determine the outcomes of single-bit transient errors at the ISA-level. Relyzer's heuristics group error sites based on similar control flow or store behavior through the notion of *equivalence classes*. In this context, an equivalence class is defined as a group of dynamic instructions with the same static PC that are expected to yield the same outcome in the presence of a bit-flip in the same register bit.

A *pilot* is a random representative from among the equivalence class. Injections performed on the pilot reflect the behavior for the entire *population*, or total members of that respective

```
Disassembly of section .text:

0000000000400cc0 <main>:
  400cc0:   41 57                 push   %r15
  400cc2:   41 56                 push   %r14
  400cc4:   41 55                 push   %r13
  400cc6:   41 54                 push   %r12
  400cc8:   41 89 fc              mov    %edi,%r12d
  400ccb:   55                    push   %rbp
  400ccc:   53                    push   %rbx
  400ccd:   48 89 f5              mov    %rsi,%rbp
  400cd0:   31 f6                 xor    %esi,%esi
  400cd2:   48 83 ec 38           sub    $0x38,%rsp
  400cd6:   48 8d 7c 24 10        lea    0x10(%rsp),%rdi
```

Figure 2.2: Example instructions of the x86 ISA.

equivalence class. All remaining members of the equivalence class are pruned as a result. Consequently, pruning becomes more effective as the population increases.

Equivalence classes fall under two categories: control equivalence and store equivalence. For both categories, a portion of the analysis requires processing *basic blocks*, or sequences of instructions separated by control instructions with a single entry and single exit point. We detail the classification procedure of each type below:

1. **Control Equivalence** - given a basic block A_1 and a control flow depth d, record the next d sequence of basic blocks, also known as a control pattern. Repeat for A_2, or the same basic block occurring later in time. If A_2 records the same sequence of basic blocks as A_1, equalize all non-store dynamic instructions in A_1 and A_2.

2. **Store Equivalence** - for a particular store instruction s_1, record its destination address and subsequent static load instructions that read from this address. Continue recording loads until either a new store instruction writes to the same address, or the program terminates. Repeat for s_2, or the same static store instruction but executed later in time. If s_2 records the same static load instructions as s_1, equalize s_1 and s_2. In addition, use store equivalence to equalize any instructions that s may depend on that are within the same basic block.

Precise techniques, such as def-use analysis and address bounding, prune based on expected error site behavior. For example, we expect that a bit-flip in a definition (def) register is equivalent to a bit-flip in its first use. Therefore, def-use analysis prunes def registers and equalizes to the error site of the first use within the same basic block. Assuming the operating system (OS) can detect memory access violations, address bounding prunes any register bit-flips that may directly cause such a violation.

Relyzer's program analysis applies all of the aforementioned techniques to prune approximately 99.99% of error sites in the best case [20]. Injection experiments are then performed on only the remaining error sites. Outcomes from Relyzer analysis are used to build a resiliency profile that highlights the distribution of error outcomes over the whole application.

2.3 ERROR OUTCOMES

There are three commonly used classifications for the potential outcomes of error injection:

1. **Masked** - The error propagation does not cause any visible effect of corruption, signifying that the output does not differ from the error-free output.

2. **Detected** - The error eventually results in either a signal to the OS that causes the program to crash, or the program hangs indefinitely.

3. **Silent Data Corruption (SDC)** - The impact of error is not visible to the user until the output is compared with the error-free output. Any deviation from the error-free output results in an SDC.

2.4 GEM5 SIMULATOR

Among the CPU simulators available, the gem5 simulator [22] has gained the most traction in the computer architecture community for three main reasons. The first is that it is fully open source. Second, active and ongoing development enables users to receive bug updates and support for new features. Third, involvement from major processor design teams (AMD and ARM) encourages collaborative efforts to develop potential industry-wide solutions [23, 24, 25].

Gem5 is an event queue based simulator; i.e., system events are handled in the order that they are received. The simulation continues until the event queue is empty. Figure 2.3 depicts behavior of the event queue processing.

An object in gem5 that can generate potential events is known as a *SimObject*. Example SimObjects include the CPU, RAM, and Disk Image. The design implication of using SimObjects enables quick configuration changes to the simulation system, such as including multiple CPUs, more memory, or custom modules. All of these configurations are declared in a user-defined script that gem5 reads at the time of startup.

Upon initialization, SimObjects may schedule their first event onto the queue, and to simulate multiple operations occurring simultaneously, a time entry is given, denoted as t_i in Figure 2.3. In this example, events scheduled in t_0 will be processed before t_1 using **ProcessEvent()**. Additionally, after processing an event in t_0, a future event may be scheduled even before all events finish in t_0 and t_1 via the **SheduleEvent()** call. The unit of time measurement used in gem5 is a *tick*, which defaults to one picosecond. The time interval $t_1 - t_0$ varies and generally depends on the system's simulated clock frequency. For example, a CPU running at 2.0 GHz uses a time interval of at least 500 ticks when scheduling instruction events.

Gem5 supports two primary simulation modes: syscall emulation (SE), or full-system (FS) simulation. In the former, there is no OS present, and it is typically used for faster runs where the involvement of an OS is not necessary. The benefit of using FS is the ability to handle signals and exceptions during application runtime. This is vital for error injection

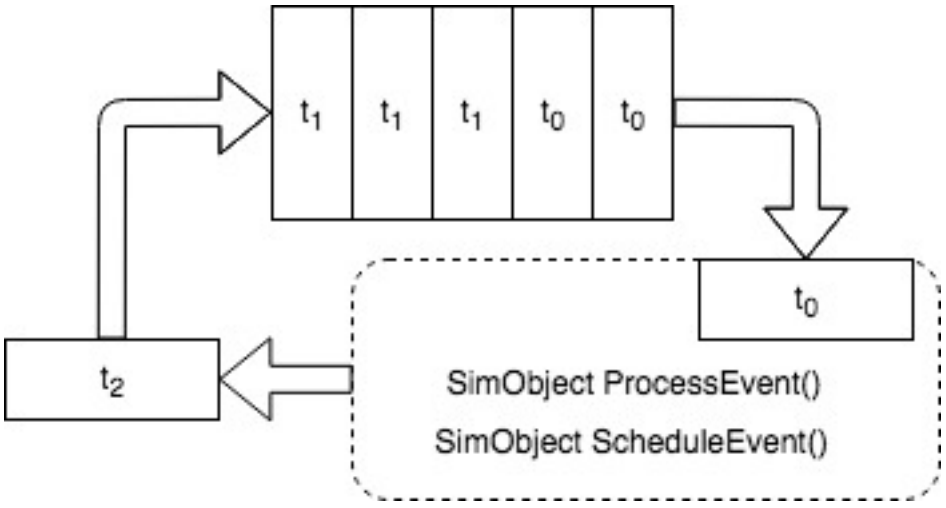

Figure 2.3: Visualization of gem5 simulator execution flow. The event-queue in the center processes events in chronological order, and SimObjects schedule new events.

experiments that detect crashes.

Complex CPU models are also available for simulation, such as InOrder and O3, which simulate in-order and out-of-order processor, respectively. A third model, Simple, assumes that memory is available immediately, therefore speeding up the simulation significantly at the cost of having inaccurate timing results. For error injection experiments where accurate timing is not required, Simple suffices.

Additionally, gem5 offers three compilation modes that impact the performance: *gem5.debug*, *gem5.opt*, and *gem5.fast*. As the name suggests, *gem5.debug* enables full debugging symbols and is primarily used to test changes made to the source code. *gem5.opt* contains fewer debugging symbols but is still able to gather trace data (explained in greater detail in Section 4). The most efficient compilation mode is *gem5.fast*, which ignores any unnecessary symbols and subsequently executes in less time than the other modes. In any of the three modes, the execution is deterministic, meaning the output of a program simulated in *gem5.fast* is always equivalent to *gem5.debug*. Checkpointing at a triggered event in the simulation is also supported in gem5. By combining checkpoints with *gem5.fast*, we are able to speedup injection experiments.

CHAPTER 3: RELYZER+

In this chapter, we overview the three phases of Relyzer+ from pre-injection to post-injection analysis. Figures 3.1, 3.3, and 3.7 aid in visualizing each phase, and each figure includes a key to distinguish which phases require gem5 simulation and the types of outputs generated.

Phase 1 profiles an application to obtain its execution and memory access trace. Then, using both the trace and supplementary information from the disassembly, Phase 2 adapts and applies each pruning technique mentioned in Section 2.2 in order to build a pruning database. With this data, Phase 3 selects pilots to perform injections on. Simulation results are then aggregated to output the final resiliency profile.

We emphasize the description of the x86 implementation in order to address the intricacies of CISC previously mentioned in Section 2.1.

The Relyzer+ implementation details discussed in this section are focused on the conceptual aspects of operating on execution traces. In addition, we provide supplemental information on the necessary gem5 functionality that one may require to further use and implement new ISAs in the future.

3.1 PHASE 1: APPLICATION PARSING AND PROFILING

In this phase, we profile an application's execution to obtain its sequence of dynamic instructions and memory accesses. Gem5 includes tracers, which are specific SimObjects used to capture this data during simulation and store the trace onto disk for future processing. Phase 1 also parses the application's disassembly in order to record the properties of each instruction such as the number of register operands.

3.1.1 Preparation

Before engaging in application profiling, the programmer must prepare each application for the simulator. Outlined below are three essential setup steps:

1. **Calculating Region of Interest** - When disassembling the compiled binary, the user must identify the start and end of the region of interest (ROI) for study for the given workload. Creating a ROI aids in filtering out phases of the application such as I/O and memory allocation that may not be relevant to the actual computation.

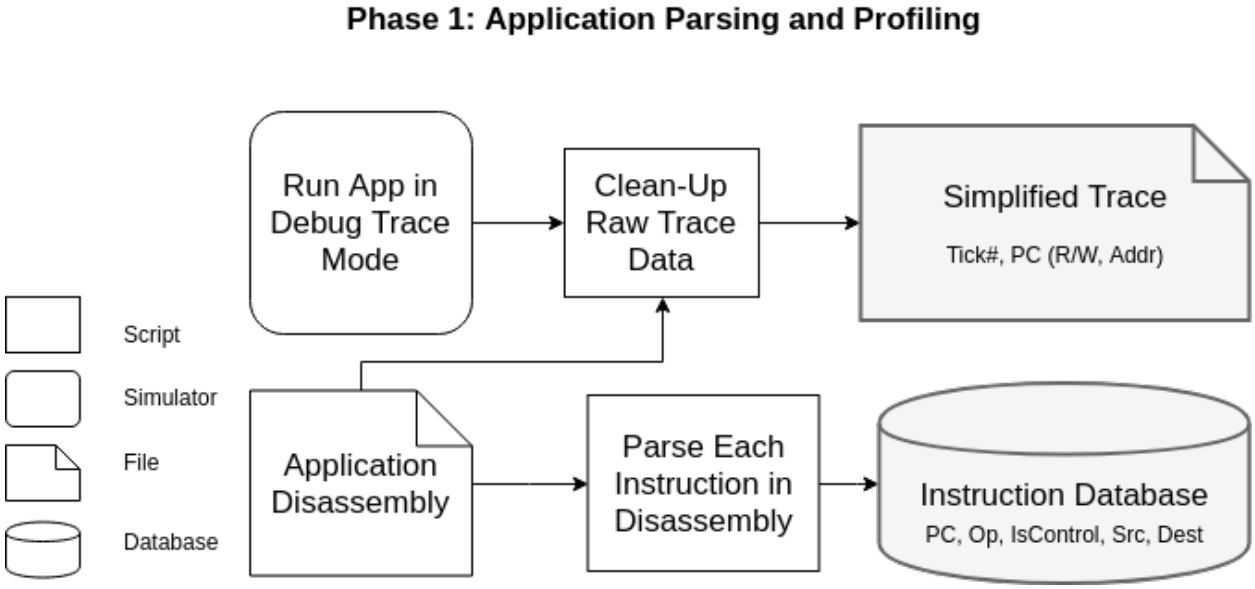

Figure 3.1: Details of the first phase of analysis, primarily generating application dynamic trace data as well as parsing the application disassembly. If any instruction accesses memory, then the simplified trace includes additional fields to indicate the appropriate memory operations. All other instructions leave these additional fields blank.

Because modern compilers aggressively optimize the assembly, finding the PCs associated with the beginning and end of the ROI may not be easily discernible. One technique is to inline dummy instructions into the source code, such as "*mov %rax, %rax*". After compilation, the user can easily find these inline instructions when examining the disassembly. The ROI should approximately begin with the first instruction immediately following the dummy instruction. The procedure of inlining assembly instructions is used when studying our applications.

2. **Capturing Output** - The output of the golden or error-free execution must be stored and parsed in a reasonable manner. By default, gem5 provides a special application binary that executes useful commands to interface between simulator and host machines. One of these commands is **writefile()**, which transfers a simulator file to the host. Depending on the corresponding host file-system in which this file is written to, a non-trivial portion of runtime may comprise of this file write.

   Alternatively, the golden run's output may be stored onto the simulated hard disk ahead of time, and the user may decide to find golden and faulty output differences within the simulator itself. The obvious drawback to this approach is that the simulator is considerably slower than native hardware at performing such a comparison. This

approach aims to remove the concern of incurring additional overhead when transferring files between simulator and host.

A third option is to create a custom script that parses the simulator's console after simulation is complete. In any of the above cases, the correctness of the experiment does not depend on how the output is stored, but merely that post-injection scripts have access to the golden and faulty outputs. In this study, we use gem5's API to transfer output from simulator to the host.

3. **Checkpointing** - If the anticipated dynamic execution trace is significantly large, then instantiating periodic checkpoints may speedup trace generation. In gem5, configuration flags enable checkpoint creation at regular intervals, or the user may decide to modify the source code and inline gem5 checkpoint calls. This allows a checkpoint to be created during irregular phases. Once the user creates the desired number of checkpoints, parallel instances of the simulation can independently collect dynamic instruction information, with the final process involving aggregation of the data. We include this potential performance enhancement into the infrastructure for evaluating larger workloads in the future.

3.1.2 Profiling

As mentioned in Section 2.4, gathering trace data requires running *gem5.opt* in order to enable the tracer SimObjects. We enable the following standard tracers provided by gem5: the instruction tracer, micro instruction tracer (exclusive for x86), and memory access tracer. The simulation incurs additional runtime overhead when inserting tracer events into the event queue for every new instruction or memory access.

Figure 3.1 highlights the stages of profiling, specifically gathering and parsing data from the tracers. Gem5 provides an option to compress the raw tracer data, but it still contains extraneous information such as SimObject names that are not relevant to the analysis. We clean the tracers by storing only the required fields and subsequently create a *simplified trace*. The precise data fields which each item in the simplified trace requires are the tick, PC, and optional fields for memory access instructions (read/write operation and the respective address). The specific tick recorded corresponds to the start of the instruction. Each tick combined with the PC uniquely identifies a dynamic instruction. In this study, we only record the PCs found within the boundaries of the disassembly. This constraint prevents analysis of library and kernel code sequences where the source code derivations may not be known.

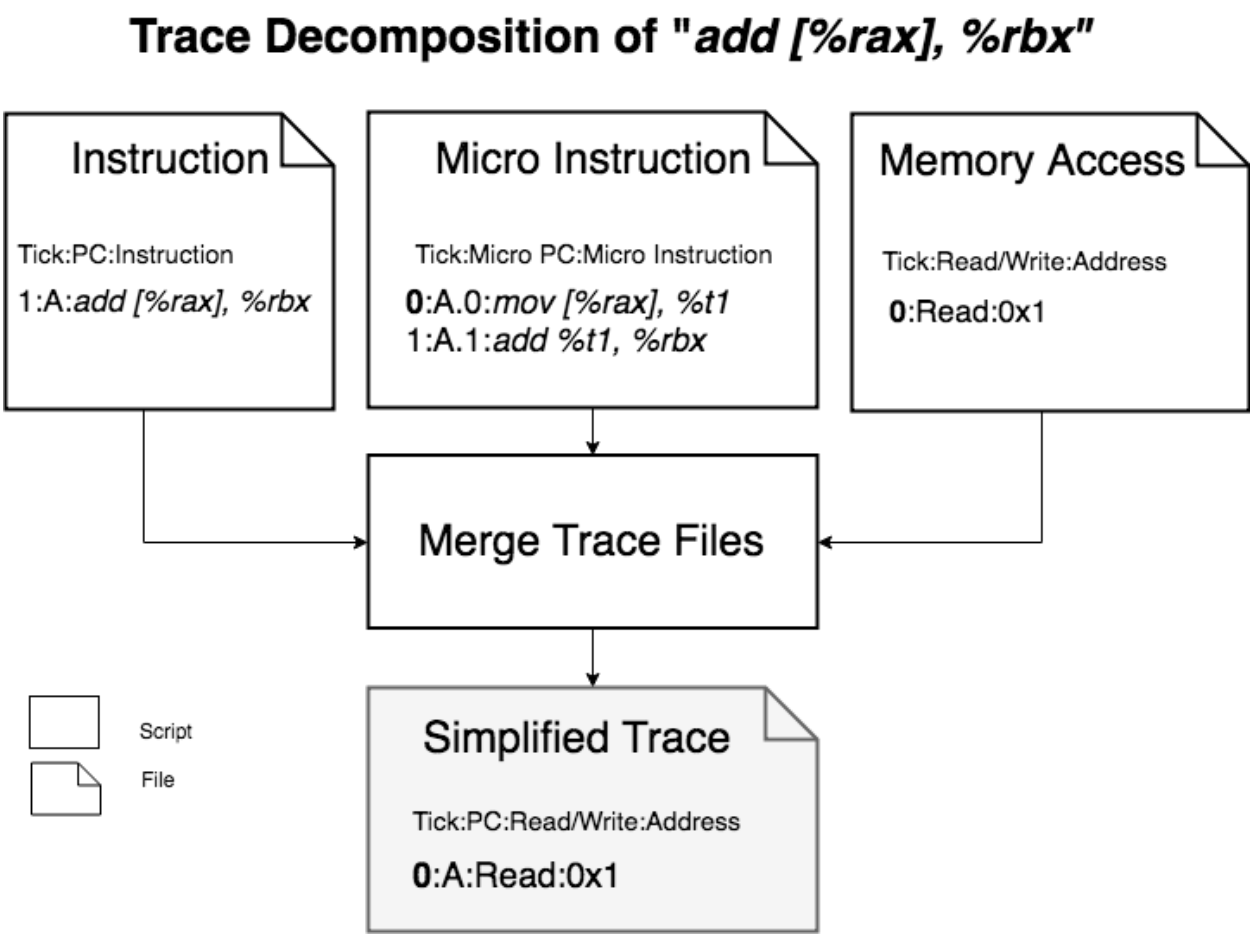

Figure 3.2: A trace decomposition of a complex instruction in x86. The final entry into the simplified trace changes PC A's tick from initially 1 to now 0, and memory access information is stored appropriately.

One of the challenges to profiling x86 applications is that gem5 decodes each ISA-level instruction into a series of micro instructions in order to address complex operations requiring more than one CPU clock cycle. The default instruction tracer only captures the tick of the *final* micro instruction. This is an issue because an instruction may begin executing several ticks beforehand, so we need to perform injections at the tick of the *first* micro instruction. To solve this problem, we enable and process a micro instruction tracer, which records the ticks of all micro instructions in the program's execution. We then find the appropriate ticks that correspond to the first micro instruction of each complex instruction.

Also, the memory trace alone does not indicate which micro instructions access memory. However, we can use a micro instruction's tick to lookup and identify if a memory access occurs at that same tick.

Figure 3.2 illustrates an example addressing and solving these issues. The complex in-

struction "*add* [*%rax*], *%rbx*" loads the value stored at address *%rax* to a temporary register *%t1* in tick 0, and then computes the addition in tick 1. When collecting the trace data, the tick recorded from the memory tracer equals the load micro instruction, or tick 0. The instruction tracer does not record any new instruction until at least tick 1. We must simulate an error injection at tick 0 instead of tick 1, but currently the trace does not indicate that behavior. Also, if we use tick 1 to search for memory accesses, we encounter the issue of being unable to find the memory read within the memory tracer.

A micro PC field is conveniently provided by the micro instruction tracer, and it stores the original PC. Referring back to the example in Figure 3.2, the micro PC for the load is $A.0$, indicating the instruction originated from PC A and is the first micro instruction. In the simplified trace, we set PC A's tick field to 0, which points to its first micro instruction. Also, we search for tick 0 and tick 1 in the memory trace and find the memory access information recorded at tick 0. We then append this information to the memory access fields for this dynamic instruction. The merged result appears in the final, simplified trace.

In general, for every instruction executed by the application, we first identify its decomposed micro instructions. The first micro instruction's tick corresponds to the beginning of the instruction's execution. In addition, for every micro instruction that accesses memory, we lookup its tick within the memory access tracer and append the corresponding information to the instruction's entry in the simplified trace.

3.1.3 Parsing Disassembly

Another important process in Phase 1 is parsing the disassembly that contains all of the application's instructions. Each ISA requires its own parser in order to create an instruction database as shown in Figure 3.1. The information stored in the database must include at least the following fields: "PC," "Op," "IsControl," "Src," and "Dest." The "IsControl" field indicates that an instruction affects control flow. The "Op" field allows for a quick lookup of information about the instruction. In x86, instructions such as *addss* (add scalar single-precision value) only affect the lower 32 bits of a floating point register, so using this field can indicate such behavior. The "Src" and "Dest" fields contain all source and destination registers respectively. As mentioned in Section 3.1.2, we do not analyze any instructions that do not belong of the application. Therefore, we also use the disassembly to identify and remove such instructions from the raw execution trace.

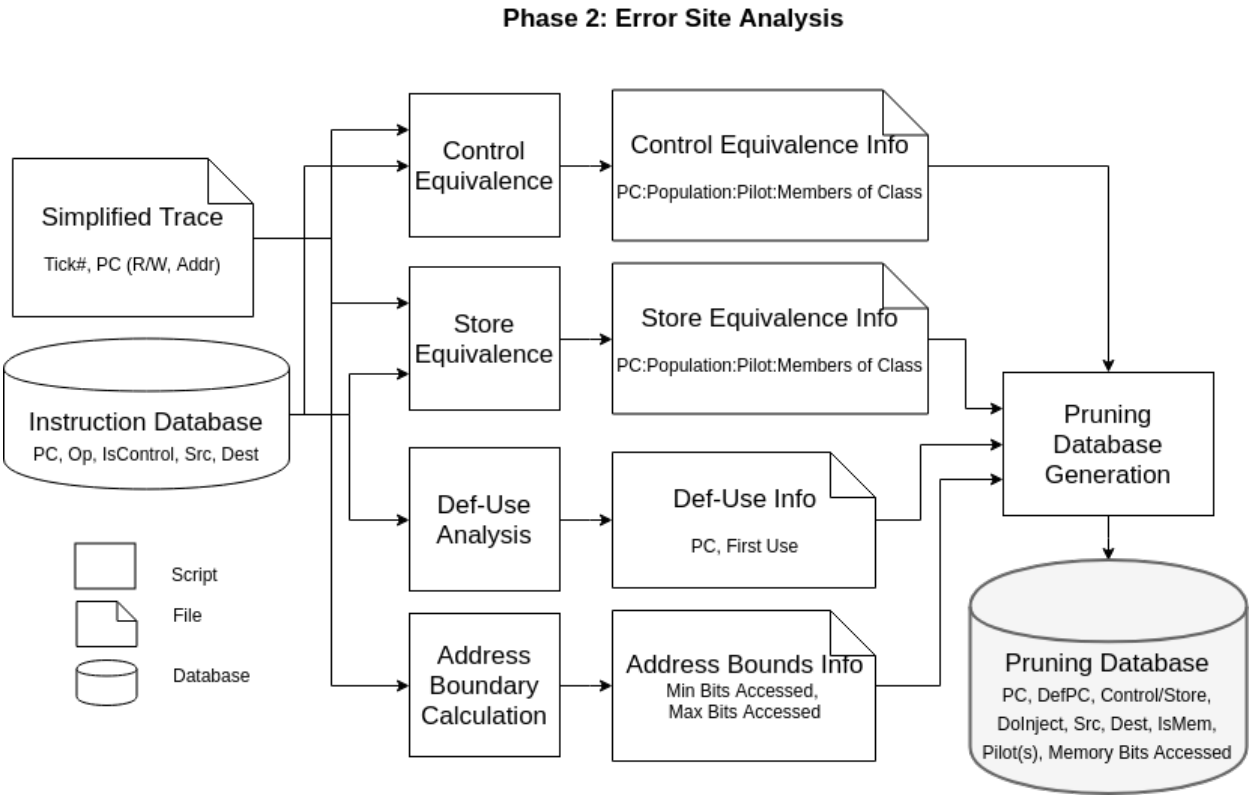

Figure 3.3: Overview of Phase 2, comprising of multiple pruning techniques. After applying all such techniques, a pruning database is generated.

3.2 PHASE 2: ERROR SITE ANALYSIS

The primary purpose of this phase is to use both the instruction database and trace data generated in Phase 1 in order to effectively prune error sites. Figure 3.3 summarizes the processes that occur in Phase 2. Also, as mentioned in Section 2.2, the key techniques are: address boundary calculation, control equivalence, store equivalence, and def-use analysis. Each technique generates output that feeds into creating the final pruning database.

3.2.1 Address Boundary Calculation

We study an application's dynamic memory profile by parsing its memory accesses found in the simplified trace. This provides us with a range of valid addresses that the application can access during runtime. We assume that errors in high order bits above the application's accessible address space are detectable by the system. Therefore, we prune all high order bits of registers accessing memory since they may result in a memory access violation. In order to effectively use this technique in a real system, the analysis should be performed on

a representative set of application inputs.

3.2.2 Contiguous Blocks

As discussed in Section 2.2, the majority of Relyzer's algorithms perform analysis at the basic block level. Because Relyzer+ analyzes an execution trace, we alter Relyzer's algorithms accordingly. To do so, we introduce the notion of *contiguous blocks*. A contiguous block is a set of contiguous instructions within a dynamic instruction trace that immediately follow a control instruction and end with a control instruction. From a static instruction perspective, there may be multiple contiguous blocks that start and end with the same PCs. For clarity, we refer to a *dynamic contiguous block* as a contiguous block that is identified based on its location in the execution trace. All of the algorithms mentioned in Section 2.2 that previously utilized basic block data structures now use contiguous blocks.

3.2.3 Contiguous Block Generation

We use both the simplified trace and instruction database from Phase 1 to create contiguous blocks. We iterate through each entry in the trace and mark control instructions. These marks identify the end of one contiguous block and the beginning of another contiguous block. Each contiguous block has its own identifier: $\{PC_{start}, PC_{end}\}$, where PC_{end} is a control instruction. As mentioned in Section 3.2.2, this identifier is not unique. Therefore, we also keep track of the tick of PC_{start}, which identifies a dynamic contiguous block. We then use the trace to build a list of dynamic contiguous blocks. Subsequently, the list of dynamic contiguous blocks is ordered chronologically; i.e., the first index in the list corresponds to the contiguous block at the beginning of execution.

3.2.4 Control Equivalence

Once the trace is transformed into a list of dynamic contiguous blocks, we form control equivalence classes based on the desired control flow depth d. To do this, we iterate through each dynamic contiguous block in the list and find the control pattern, or sequence of d subsequent contiguous blocks that the application traverses over. Because the list of contiguous blocks is sorted chronologically, the control pattern is simply a sub-list starting from the current index of the dynamic contiguous block to the next d indices. The control pattern is then compared with previous patterns to ensure it was not observed previously. If this is the case,

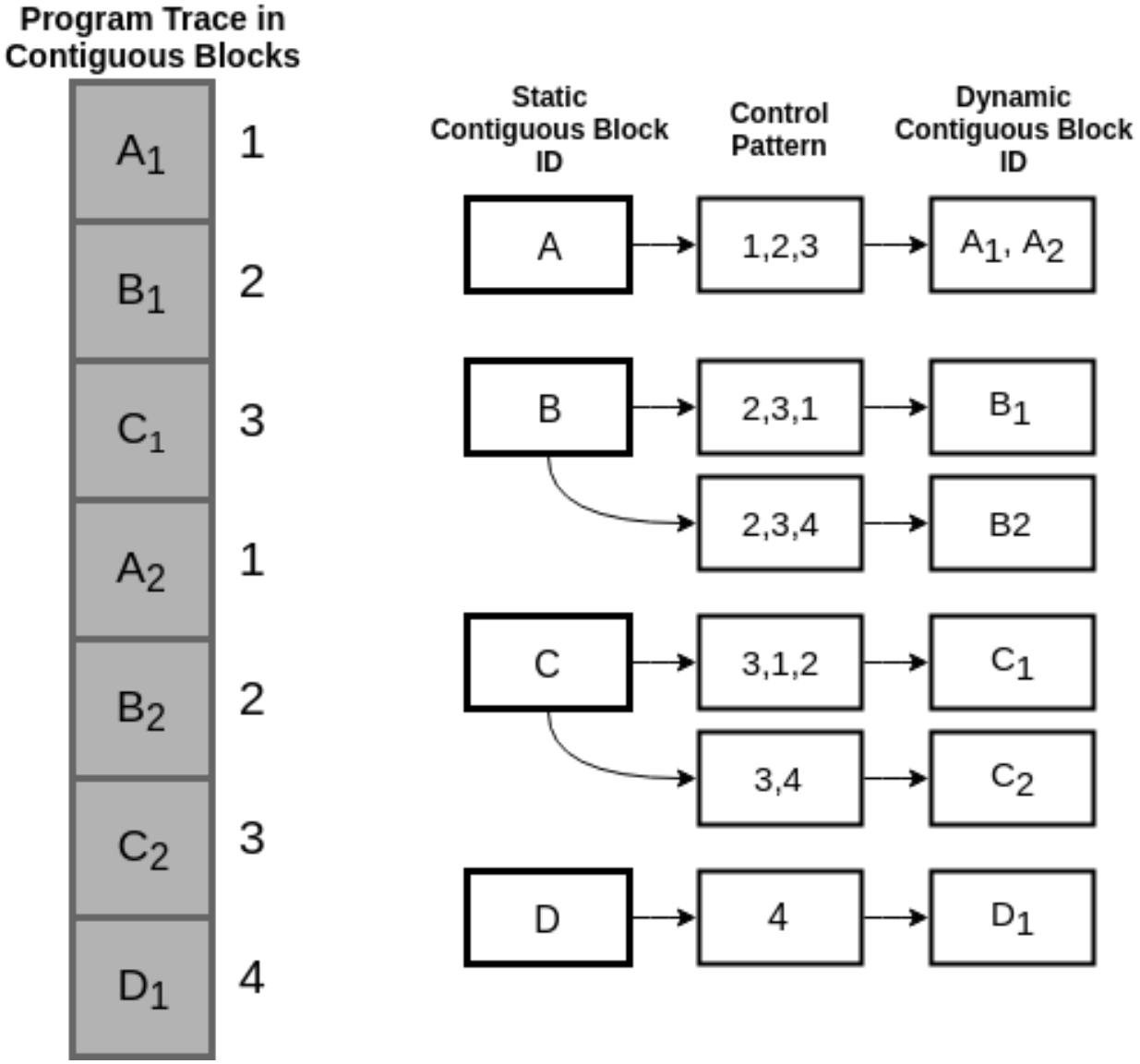

Figure 3.4: Organization of the data structures involved with control equivalence class generation, with a maximum control depth of 3 as an example. Whenever control patterns for two or more contiguous blocks match, they are equalized. A_1 and A_2 are the only contiguous blocks in this example where this is the case.

then a new equivalence class forms. Otherwise, we equalize the current dynamic contiguous block with the other dynamic contiguous blocks observing the same control pattern.

Figure 3.4 summarizes the aforementioned formation of control equivalence classes at the contiguous block level. In the example, both A_1 and A_2 encounter the same control pattern of $[1, 2, 3]$, so we equalize A_1 and A_2. In the case of B, two different control patterns are encountered, so B_1 and B_2 are not equalized.

One factor to consider when employing control equivalence is that the number of equivalence classes per contiguous block may increase dramatically if many different control patterns are observed at depth d. This is problematic for two reasons: A) the space required to store the observed control patterns grows, and B) the pruning effectiveness decreases since each new equivalence class adds an injection location. A method to combat the growth is to dynamically reduce d. To do this, we define an equivalence class capacity c as a constraint. If the number of current equivalence classes N for a given contiguous block is greater than c, reduce d until $N \leq c$. When reducing d, all dynamic contiguous blocks need to be re-equalized accordingly.

Once all equivalence classes are formed, we randomly select one pilot out of the members of an equivalence class for each equivalence class and store for later parsing (Section 3.2.7).

3.2.5 Store Equivalence

Store equivalence consists of two parts. One part of store equivalence involves determining the instructions each store instruction depends on. Again, we limit ourselves to analyzing at the contiguous block level, so we reuse the program's contiguous blocks generated previously. To identify all instructions the store depends on, we begin with the store instruction and traverse the contiguous block backwards. The first instruction a that defines the store instruction's source register is added to a list of instructions that this store depends on. The chain of depending instructions continues from there, as instruction a has its own source registers that depend on other instructions in the contiguous block. This process continues until either the beginning of the contiguous block is reached, or there are no more instructions that the store depends on. When building such dependence chains, an additional lookup occurs for x86 that ensures that the def register names do in fact alias to the same source register.

The second part of store equivalence equalizes multiple dynamic invocations of a given static store instruction based on the subsequent loads observed. To do this, we filter the execution trace to include only load and store instructions. Then, we apply the store equivalence classification procedure from Section 2.2. Figure 3.5 exemplifies this procedure. Store

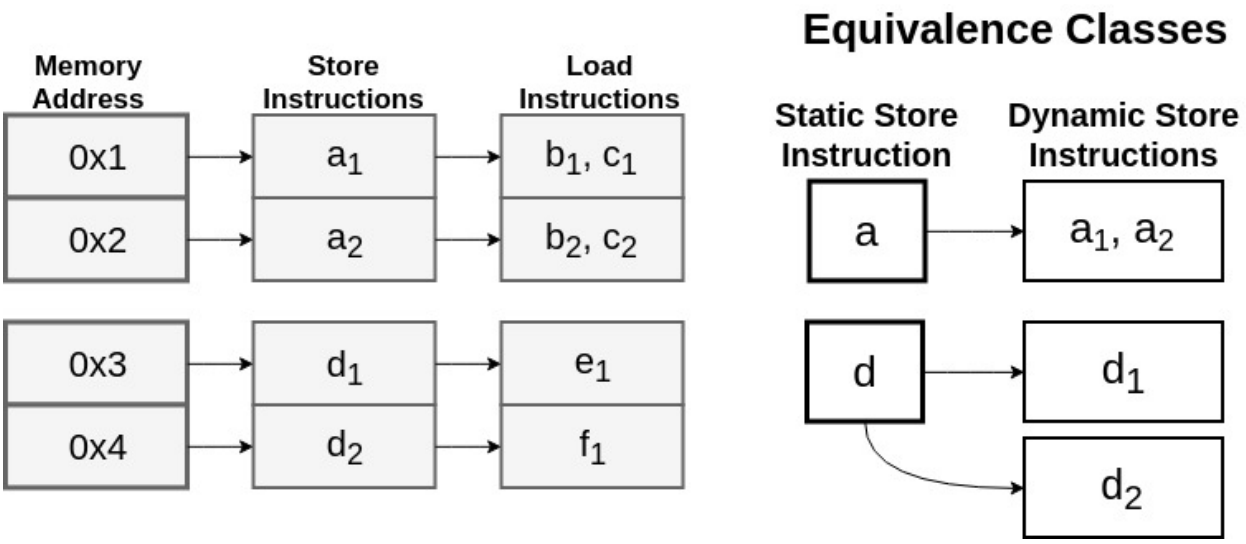

Figure 3.5: Organization of the data structures involved with store equivalence class generation. For two or more dynamic stores, if the list of subsequent static loads are the same, then the stores are equalized. In this example, a_1 and a_2 are equalized because they both encounter loads b, c. In contrast, d_1 and d_2 are not equalized because d_1 encounters e, and d_2 encounters f.

instructions a_1 and a_2 are within the same equivalence class since they both share the same subsequent static loads: b and c. However, store instructions d_1 and d_2 are binned into different equivalence classes because the subsequent load instructions are not identical.

Following the same procedure as control equivalence, we randomly select one pilot out of the members from an equivalence class.

3.2.6 Def-Use Analysis

As mentioned in Section 3.2.2, we confine def-use analysis to the contiguous block level. Consequently, the generated contiguous block data structures are also used for this technique.

The standard procedure for pruning def registers described in Section 2.2 requires modification in order to properly analyze x86 registers. Specifically, registers that have a size ranging from 8-32 bits alias to the lower bits of a 64-bit register, therefore, accessing only a portion of the register. As a result, *partial def-use pairs* can now form across registers of different sizes. In order to capture the partial def-use pairs, we design two databases: a Contiguous Block Def Register Database and an Instruction Def Register Database.

Figure 3.6 provides an example of how each instruction accesses the data structures to address partial def-use pairs. The procedure for modifying the databases in this example is as follows:

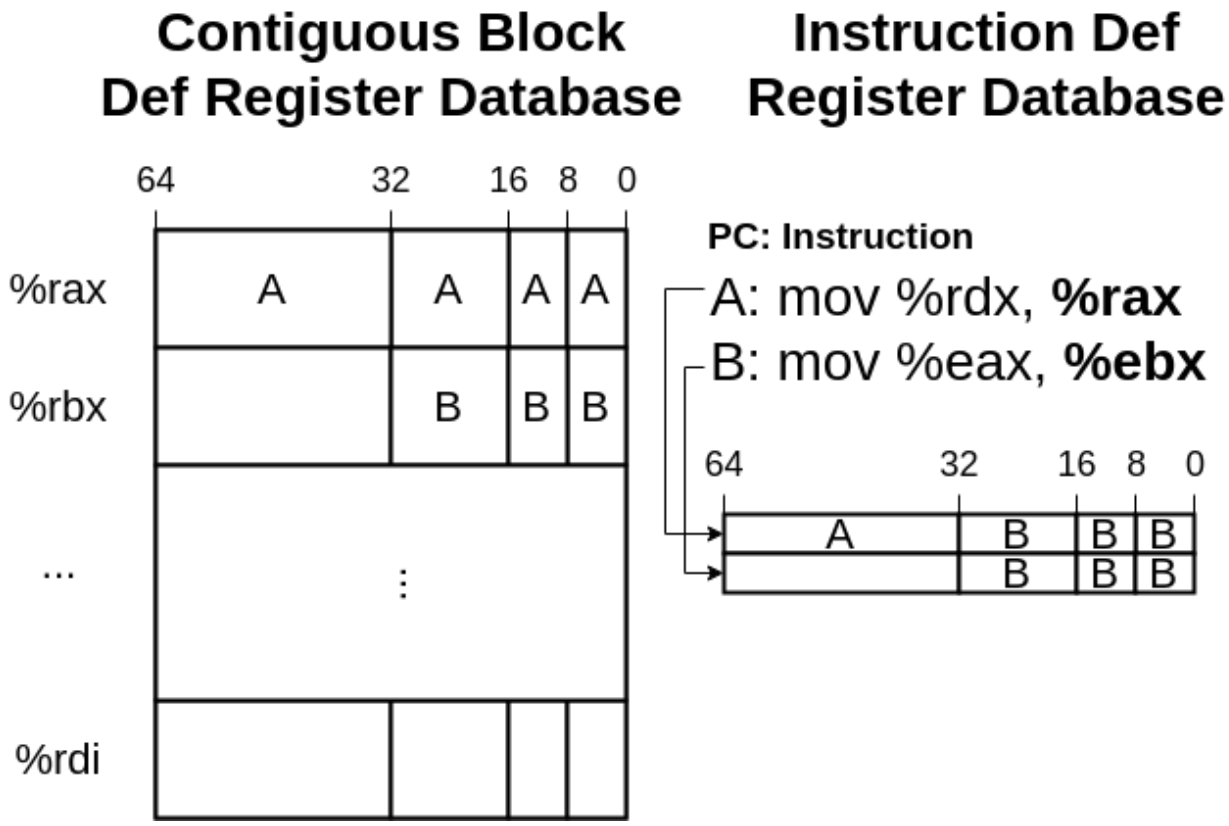

Figure 3.6: Organization of the data structures involved with def-use analysis. PC A has *%rax* as its def register, and PC B first uses the lower 32 bits of *%rax* via *%eax*. Whenever a register is last defined, all appropriate fields within the Continuous Block Def Register Database are filled with the corresponding PC. The Instruction Def Register Database stores the bit fields that may be modified after a first use. This is shown in PC A's entry in the database, where PC B now occupies the lower bit fields.

1. Initially, the Contiguous Block Def Register Database is empty. When PC A is analyzed, we encounter a def of *%rax*. This is a 64-bit register, so we fill all fields in the Contiguous Block Def Register Database for entry *%rax* with PC A.

2. Each PC also has its own entry in the Instruction Def Register Database. Again, *%rax* uses all 64 bits, so all fields for entry PC A are initialized to PC A.

3. For each source register in PC B, we identify the 64-bit register alias. *%eax* aliases to *%rax* in this case. The entry for *%rax* is checked within the Contiguous Block Def Register Database.

4. We begin traversing the fields of entry *%rax* in the Contiguous Block Def Register Database starting from the least significant bits to the most significant bits of the

original source register, *%eax*. Therefore, we only access the lower 32-bit fields of entry *%rax*. Each field that is not empty points to the def PC that last defined this bit range. We go to the same field in this def PC's entry within the Instruction Def Register Database. If that corresponding field is equal to the original PC (A in this case), then it has not been previously used. We replace this field with the current PC, which is B. As a result, we correctly prune the lower 32 bits of *%rax* from PC A. In general, any field that is later modified within the entry of the Instruction Def Register Database indicates that the corresponding bit ranges are to be pruned.

Once we encounter a new contiguous block, we clear all PCs in the Contiguous Block Def Register and repeat the process. The result of using these databases is that def-use specifically prunes only the number of bits that are actually used.

3.2.7 Combining Analysis Files

Once all of the relevant analysis scripts complete, the resulting files then feed into a pruning database generation module. The newly-formed pruning database provides enough information to actually create the error injection campaign. The fields for the database are displayed in Figure 3.3. We include a "DoInject" field in the pruning database to indicate whether any registers need injections after pruning. If this flag is false, then we can skip over this instruction. The "DefPC" field is relevant for mapping the PCs that first use the def register (if applicable). In x86, this field also highlights which specific bits are pruned via def-use and which still require injections. The "IsMem" field indicates that the instruction contains registers accessing memory, so bits outside of the address bounds are to be pruned. We use the "Control/Store" field to determine which equivalence class pruning technique to use. By default, all store instructions and the instructions that the store depends on are marked to select pilots using store equivalence. The remainder of instructions are marked to select pilots using control equivalence.

3.3 PHASE 3: ERROR INJECTION EXPERIMENTS

The last phase of the error injection framework, depicted in Figure 3.7, creates injection experiments and performs the simulated injections. The pruning database, combined with an ISA-dependent script, generates the list of injections to be performed. We iterate through every instruction's registers within the pruning database to create injections for all equivalence classes.

Identifying the registers and their corresponding lengths requires a user to provide the proper ISA register decoder. This necessity becomes more apparent when calculating the precise bits to inject into. For example, if the target register is only 32 bits in length, and the address boundaries occur in the first 40 bits, then we take the minimum of the two lengths and inject only into those bits, again assuming the user decodes the register length correctly.

The injection list contains all remaining error sites to perform injections upon completion of Phase 2. Section 3.5 describes the gem5-related operations to perform injections. Even after pruning, if a program's injection list is too large to complete in a reasonable time, then an alternative is to select only injections that account for the top $X\%$ of population among all equivalence classes. In prior work, X ranged from 95% to 99% and still produced tangible results [16, 20], but by not injecting into 100% of the population, the final outcome may miss some SDCs. None of the injection lists in this study resulted in adjusting X, so we explore injections that account for 100% of the population.

All injection experiments eventually aggregate to form one raw injection results file. We use the pruning database once more to identify the outcome of the pruned def register bits based on the injection of the first use. After determining the outcome, we insert the pruned def register bits as well as the out-of-bound address bits into the results file. To simplify this file, we categorize the results over all static instructions by identifying the worst case outcome for each static instruction. The outcomes from best to worst case are: Masked, Detected, and SDC. Relyzer+ outputs this resiliency profile after completing all phases.

3.4 ADDITIONAL ISA CONSIDERATIONS

Each ISA has its own design details that may slightly modify the algorithm. Some examples are listed below:

SPARC contains a branch delay slot to improve pipeline performance, so the compiler often inserts or reorders instructions to fill in this slot. As a result, the actual sequence of execution may appear out of order at a branch instruction. The delay slot affects contiguous block generation, which now must account for another instruction following the branch when computing the end of a contiguous block.

There are also certain condition code registers in SPARC such as %icc that are modified as a side effect of other instructions. The ISA uses these registers to resolve branches. SPARC includes internal registers that are not visible to the user such as the PSTATE register. To simplify our analysis, we do not analyze condition code registers nor internal registers.

We already identified several x86 design challenges, and the following are more assumptions to simplify the analysis. Registers used for SSE instructions, such as *%xmm0* are 128 bits

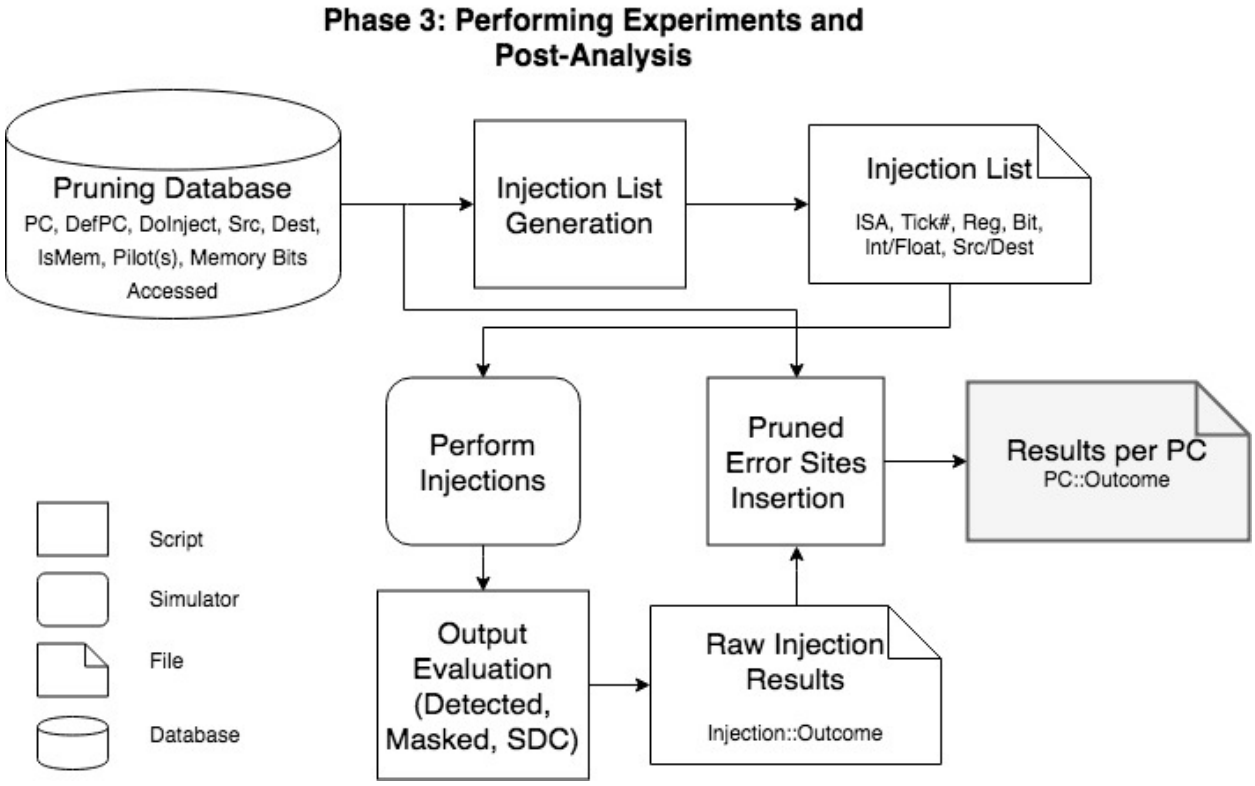

Figure 3.7: Overview of phase 3. Actual injection simulation, collection of outcomes, and post-injection analysis occurs in this phase.

wide. However, in our study, we focus our analysis on the lower 64 bits because double-precision floating point computation only affects these bits.

Like SPARC, x86 contains condition code registers such as *RFLAGS* and internal registers such as *%cr0* that we omit from analysis. To simplify further, we do not perform bit-flips in registers that directly modify the stack such as *%rbp* and *%rsp*. However, instructions such as *push* and *pop* are valid, provided that the register being pushed onto or popped from the stack is not one of the above registers.

3.5 ERROR INJECTION IN GEM5

To actually inject errors into the simulation, an injector module attaches to the CPU and performs bit-flips at the appropriate time in simulation. The injector is a SimObject, so instantiating its parameters relies on modifying the full system configuration script as mentioned in Section 2.4. We create additional parameters for the configuration script that enable error injection at a particular tick, register, bit, and timeout value. A user now can simultaneously launch multiple configurations, therefore either simulating injections into

different error sites or injecting into different applications altogether.

To actually flip a bit of a particular register, we leverage the provided gem5 API functions to read or write register values. We read the initial register value using **getIntRegValue()** or **getFloatRegBits()**, and then flip the appropriate bit. We then set the register to its faulty value using **setIntRegValue()** or **setFloatRegBits()**. If the register is a source, then we perform the injection immediately. Otherwise, we proceed to the starting tick of the next instruction and perform the injection at time. This is to ensure that the destination register updates its value upon completion of the instruction.

The simulation runs to completion, and we store the final output. We then proceed to bin the outcomes into the appropriate categories mentioned in Section 2.3. The simplest way for categorizing the injection outcome as SDC is that the final output file differs from the golden output file. On the other hand, if the files are identical, then we categorize the outcome as Masked. We check the two files by using a standard diff utility.

There are several detectors used to bin outcomes as Detected. The most frequently used are those that check for segmentation faults, exceptions, and error codes. The injector SimObject contains a built-in timeout detector. We set its value such that conservatively if the faulty run has executed more than twice the amount of ticks logged in the golden execution, a timeout is triggered. Currently all detectors aside from the timeout detector parse the simulator console to find detection symptoms, but in the future we plan to also develop the appropriate SimObjects to further speedup the outcome binning process.

CHAPTER 4: EVALUATION METHODOLOGY

This chapter outlines the necessary assumptions that we make when performing the analysis. We describe our error model, validation procedure, and workloads.

4.1 ERROR MODEL

This study focuses on analyzing single-bit transient errors in the integer and floating point registers of dynamic instructions. As mentioned in Section 3.4, we do not inject errors into condition codes registers and internal registers that are not directly accessible to the user. We also do not inject into registers directly affecting the stack, such as *%rsp* and *%rbp*. However, we still inject into instructions that implicitly modify these registers, such as *push* or *pop*, provided that neither *%rsp* nor *%rbp* are being pushed onto or popped from the stack.

4.2 VALIDATING HEURISTICS

So far, the heuristics are not supported with any level of confidence that the pilots selected actually reflect error behavior of the entire population. Therefore, validation of both control and store equivalence heuristics is crucial. If either heuristic does not exhibit high confidence, then the remainder of the results lose significance because the pilot selected for injections does not accurately account for the entire equivalence class. To perform validation, we inject errors into other members from within the equivalence class and compare the outcomes to the original pilot's outcome.

Validating that every single member of an equivalence class will result in the same outcome is expensive. Instead, we randomly sample members until we reach high confidence. We also take a similar approach employed by previous work [20] by injecting into every 8th bit of a register instead of iterating each bit sequentially to reduce the number of experiments further. To attain a 95% confidence interval with an error margin of 5%, we sample up to 400 members from within an equivalence class. We then continue to sample additional equivalence classes with the limitation of approximately 200,000 injections per control and store equivalence (due to simulation time constraints) for each application. Aggregating the validation statistics for each application involves first identifying the original pilot outcome per equivalence class. Then, we determine the outcome of the other sampled members. We keep track of each outcome that correctly matches with the pilot's original outcome, then

Suite	Application	Domain	Input
Parsec 3.0 [26]	Blackscholes	Financial Modeling	21 options
	Swaptions		1 option 1 simulation
SPLASH-2 [27]	LU	Scientific Computing	16x16 matrix 8x8 block size
	FFT	Signal Processing	2^8 data points
ACCEPT [28]	Sobel	Image Processing	81x121 pixel image

Table 4.1: Applications studied across a variety of domains and suites.

divide by the total population sampled to compute the accuracy of the equivalence class. Finally, we calculate a weighted average of the accuracy based on the population over all sampled equivalence classes. Using a weighted average ensures that larger equivalence classes contribute more towards overall accuracy.

4.3 WORKLOADS

We select workloads from the Parsec 3.0 [26], SPLASH-2 [27], and ACCEPT [28] benchmark suite. Table 4.1 summarizes each application, domain and input size. We set the value for control equivalence capacity $c = 50$ and initial control pattern depth $d = 50$ for all applications. By using similar workloads from prior work [29], we carefully reduce the size of the input in order to speedup injection runs but still capture each application's relevant execution paths. For SPARC workloads, we reuse the memory profile information collected from prior work [20] in order to calculate each application's address boundaries (Section 3.2.1). This affects SPARC results when comparing with x86 in Section **??**, and understanding the memory access behavior between ISAs is future work. Potential storage issues mentioned in Chapter 3 were not observed in any of the workloads.

4.4 EXPERIMENTAL INFRASTRUCTURE

We use GCC [30] to compile x86 binaries and Solaris CC [31] to compile SPARC binaries. In both cases, full optimizations (-O3) were enabled. We simulate x86 applications using gem5 in FS mode with the Simple CPU model. SPARC applications are simulated on a similar system but use Wind River Simics[21] as the simulation platform. In this study, a higher number of processors is beneficial since each injection simulation operates independently. We take advantage of this parallelism by running error injection simulations on a

200-node cluster of 2.4GHz Intel Xeon E7-8870 processors with 252GB of total memory.

CHAPTER 5: RESULTS

This chapter describes preliminary results from evaluation. In this study, we compare Relyzer+'s pruning effectiveness across x86 and SPARC. We also perform validation experiments for x86. The chapter concludes by describing the outcomes of error injection experiments across both ISAs.

In this study, SPARC results differ from previous work [20] because as mentioned in Section 4.3, we reduce the input size of each workload.

Application	**ISA**	**Initial Error Sites**	**Total Pruning**	**Remaining Error Sites**
Blackscholes	x86	21.8M	99.51%	107,308
	SPARC	762,656	52.62%	361,349
FFT	x86	7.1M	95.65%	309,469
	SPARC	10.3M	94.34%	584,854
LU	x86	4.2M	86.94%	551,266
	SPARC	1.9M	71.96%	535,802
Swaptions	x86	13.3M	93.73%	833,875
	SPARC	23.6M	94.40%	1.3M
Sobel	x86	183.2M	99.85%	273,289
	SPARC	1.1B	99.97%	293,632

Table 5.1: Number of error sites pruned per application.

5.1 PRUNING EFFECTIVENESS

Table 5.1 displays pruning effectiveness for both ISAs, and Figure 5.1 provides a detailed breakdown of each technique's contribution to pruning. Aside from Blackscholes-SPARC, the total error sites being pruned is significant for both ISAs. On average, pruning effectiveness is 95.1% and 82.7% for x86 and SPARC, respectively. One possible explanation for the reduced effectiveness in Blackscholes-SPARC is that the number of error sites is relatively small compared to the other applications at 763,000. In fact, Blackscholes-SPARC only executes 7153 dynamic instructions within its ROI. In general, control equivalence relies on larger execution sizes with the assumption that the number of loops executed increases, thereby equalizing more instructions. As expected, because of Blackscholes-SPARC's small

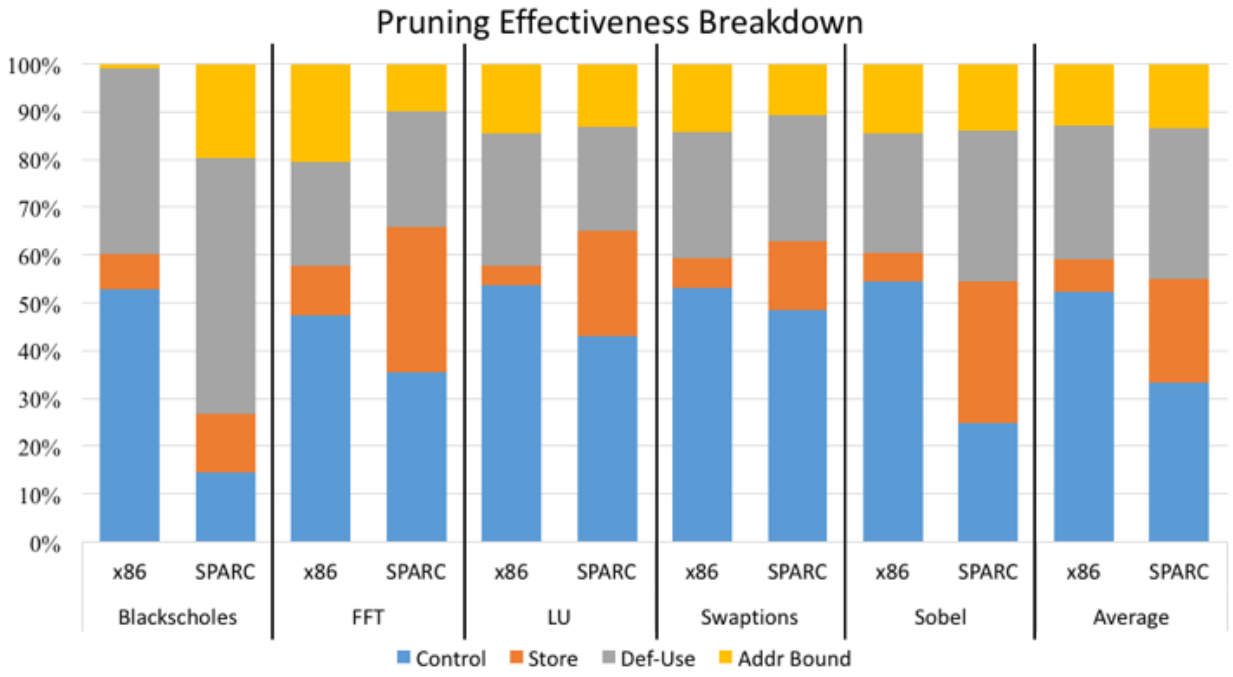

Figure 5.1: Pruning effectiveness for each ISA.

execution size, control equivalence's pruning contribution is only 14.4%. With the exception of LU-x86, there are also fewer remaining error sites for x86, and the difference ranges from only 20,000 in Sobel to 485,000 in Swaptions.

x86 pruning effectiveness primarily comprises of control equivalence, which prunes 52.3% of all error sites on average. Store equivalence prunes more for SPARC than x86 in the average case at 21.8%. Analyzing the compiled binaries may lead to understanding the pruning technique contributions between the two ISAs, and we leave that for future work.

5.2 VALIDATION

Figure 5.2 reports the validation accuracy per application. For every workload, outcomes of different members of the equivalence classes almost always match the outcome from the original pilot selected. This is evident from the high average accuracy of 97.7% for store equivalence and 98.2% for control equivalence. There were some equivalence classes where accuracy was near 0%. This scenario happens infrequently, however, at 0.06% of all equivalence classes analyzed. Recall from Section 2.2 that a pilot is a randomly selected member from the equivalence class. There is a chance that the pilot does not reflect the majority of members within the equivalence class. Returning to the near-0% accuracy equivalence

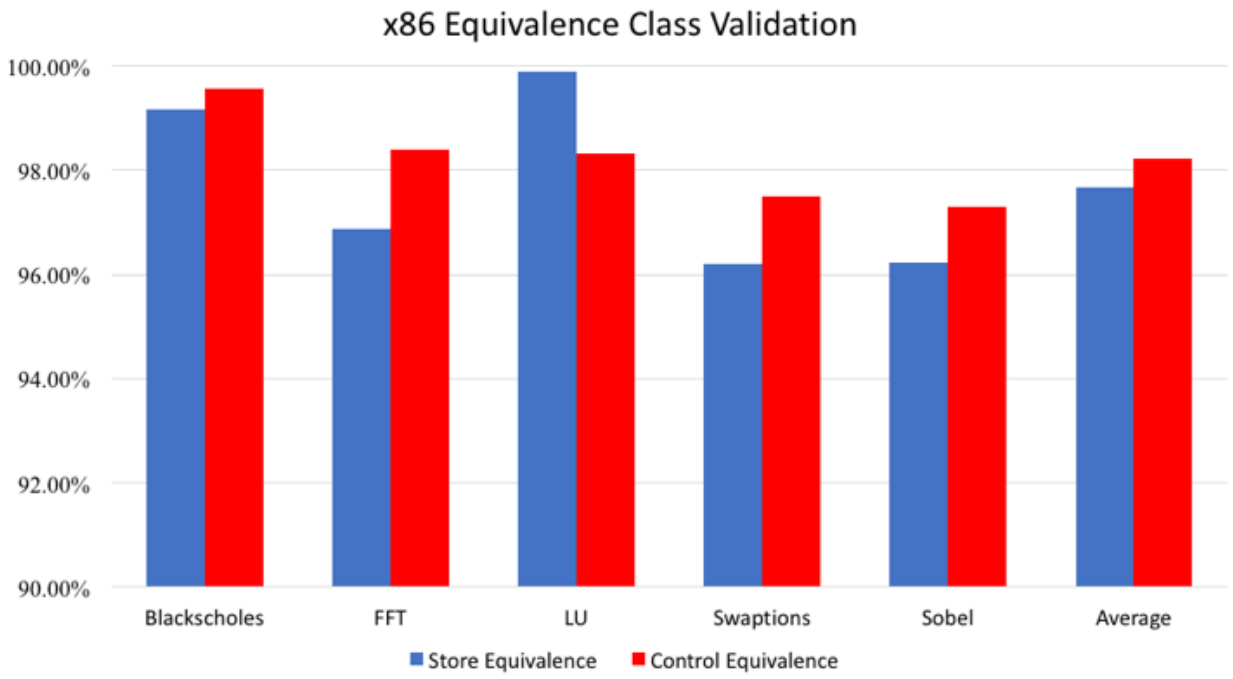

Figure 5.2: x86 validation results for both control and store equivalence classes at 95% confidence interval with 5% error margin. In both categories, accuracy is above 96%.

classes, if we select a different pilot and rerun validation experiments, then the accuracy will significantly improve.

5.3 ERROR OUTCOMES

When analyzing Relyzer+'s error outcomes, determining a proper comparison point between RISC and CISC ISAs is challenging. The difficulties emerge from differences in instruction semantics, number of registers used per instruction, and number of instructions, both static and dynamic. Subsequently, we opt to not report such statistics because of the lack of a method to compare these values across ISAs fairly. Instead, we choose to equalize with regards to the distribution of each application's error outcomes (Masked, Detected, or SDC). An advantage here is that although the absolute number of instructions and total injections differs, from a relative standpoint, the distribution of error outcomes is directly comparable.

We report the error outcomes in Figure 5.3. As mentioned in Section 3.3, we identify the worst case outcome per PC as SDC, followed by Detected, and finally Masked. For both ISAs, on average, more than 60% of PCs uncover SDCs. In principle, uncovering a high

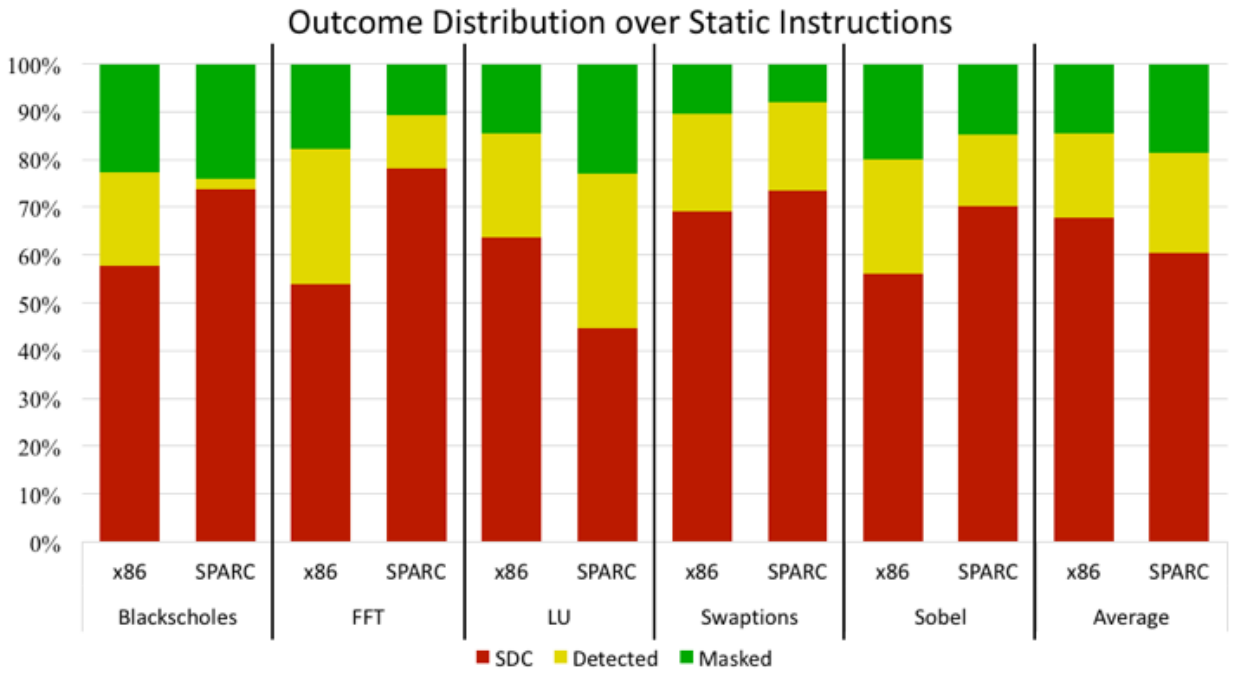

Figure 5.3: Distribution of outcomes for each PC based on the worst case injection outcome.

percentage of SDC-causing instructions indicates that the tool is effective at finding such instructions. However, this does not take into account the number of dynamic invocations of each PC. Based on the input size, the majority of the SDC PCs may only comprise a small fraction of the total execution. Because the size of the input may not be known ahead of time, we do not report on a dynamic instruction basis.

On average, 67.8% of static instructions yield SDCs for x86. Similarly, SPARC results display a slightly reduced average percentage of SDC PCs at 60.4%. On the other hand, all x86 applications except for x86-LU uncover fewer SDC-causing instructions. Further analysis is required to understand the cause for such differences between ISAs.

CHAPTER 6: RELATED WORK

We already mentioned Relyzer in Section 2.2, and the majority of related work also performs error injection simulation experiments. Trade-offs between either accuracy, robustness, or usability are distinguishing features for each contribution. In this chapter, we compare and contrast various approaches to error injection.

Several recent tools also build upon gem5 as their base simulator, specifically MeRLiN [15] which utilizes GeFIN [32] to simulate micro-architectural injections in an x86 O3 CPU. Another error injection tool that operates at the micro-architectural level and also supports both Alpha and x86 is GemFI [33]. This tool is open source, whereas GeFIN is not at the time of publication. One primary drawback for these micro-architectural injection tools is that as a CPU's micro-architecture changes, new injection experiments must be performed that accurately reflect those changes.

Other error injection tools, such as LLFI [34], analyze applications at the intermediate representation (IR) level. IR is ISA-independent by design, so such an analysis would ideally hold true regardless of the hardware architecture. However, there may be loss in error site accuracy because IR still requires additional transformations before producing actual assembly [35].

Finally, like Relyzer, there are related works that perform ISA-level analysis. FAIL* [17], GangES [18], and Approxilyzer [16] are notable examples. A benefit to performing ISA-level injections is that the results provide instruction-level resiliency information. System designers can then utilize these results to create soft error protection schemes at the instruction-level [36, 37]. FAIL* also uses gem5 and supports ARM but is limited to one pruning technique: def-use analysis.

GangES and Approxilyzer are both works that extend Relyzer analysis. GangES groups injection simulation experiments and periodically compares the simulator state after injecting the error. This check allows for early injection termination if equivalent state is found, which speeds up Relyzer. GangES' performance enhancements may also find its way into our work in the future.

Approxilyzer [16] classifies injection outcomes in a similar matter to Relyzer except that SDCs are further binned based on the user-provided quality metric: SDC-Good, SDC-Maybe, SDC-Bad, and DDC (Detectable Data Corruptions). This new spectrum indicates a varying level of quality degradation in the final output. Precise classification of output corruptions informs users that some regions of execution can be approximated. Performing an Approxilyzer-level analysis is also supplemental to this work and adds the number of ap-

proximable instructions as another comparison point between ISAs in future work.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

7.1 CONCLUSIONS

This work proposes key concepts to bring about a new error injection tool capable of providing comprehensive analysis across more ISAs. By leveraging the open source gem5 simulator, we develop Relyzer+, which adapts Relyzer's initial algorithms to operate on an execution trace and supports x86. An analysis of Relyzer+'s equivalence class heuristics on x86 indicates a high confidence that pruning is both effective and accurate, with average accuracy above 96%. We also perform a preliminary comparison between x86 and SPARC. Applications from both ISAs use Relyzer+'s pruning techniques to drastically reduce the number of error sites. After performing error injection experiments, our results show that different levels of SDCs were uncovered across each application. On average, 68% of static instructions yield SDCs on x86. The percentage of static instructions yielding SDCs for SPARC averages to 60%. We only establish a few comparison points in this work, and more detailed evaluations are necessary to understand an ISA's role in overall resiliency.

7.2 FUTURE WORK

With the provision of an open-source framework, any user can explore several new paths in the domain of reliability and approximate computing. An intuitive first step is to develop support of popular ISAs such as ARM. Performing an in-depth comparison between ISAs is another direction that can reveal new insights to an architecture's impact on resiliency. Incorporating resiliency analysis into the software development workflow is another avenue that if proven useful, could have monumental impact, especially as software continues to evolve. We intend to leverage gem5 and its existing multi-core CPU and GPU simulation features in order to evaluate parallel workloads. Overall, Relyzer+ facilitates future work in any of the aforementioned directions.

REFERENCES

[1] S. Borkar, "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation," *IEEE Micro*, vol. 25, no. 6, 2005.

[2] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 389–398.

[3] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[4] 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 *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, Dec 2003, pp. 29–40.

[5] T. J. Dell, "A White Paper on the Benefits of Chipkill - Correct ECC for PC Server Main Memory," in *IBM*, 1997.

[6] B. Randell, "System Structure for Software Fault Tolerance," in *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 437–449.

[7] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, June 2016, pp. 168–179.

[8] S. Sahoo, M.-L. Li, P. Ramchandran, S. V. Adve, V. Adve, and Y. Zhou, "Using Likely Program Invariants to Detect Hardware Errors," in *Proc. of International Conference on Dependable Systems and Networks*, 2008.

[9] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer, "Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware," in *Proc. of European Dependable Computing Conference*, 2006.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, Feb 2001.

[11] H. J. Wunderlich, C. Braun, and A. Schll, "Pushing the limits: How fault tolerance extends the scope of approximate computing," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, July 2016, pp. 133–136.

[12] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Test Symposium (ETS), 2013 18th IEEE European.* IEEE, 2013, pp. 1–6.

[13] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation Test in Europe Conference Exhibition*, April.

[14] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, "Statistical Fault Injection," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, June 2008, pp. 122–127.

[15] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17, 2017.

[16] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a Systematic Framework for Instruction-level Approximate Computing and its Application to Hardware Resiliency," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–14.

[17] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance," in *Dependable Computing Conference (EDCC), 2015 Eleventh European*, Sept 2015, pp. 245–255.

[18] S. K. Sastry Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "GangES: Gang Error Simulation for Hardware Resiliency Evaluation," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 61–72.

[19] J. Li and Q. Tan, "SmartInjector: Exploiting Intelligent Fault Injection for SDC Rate Analysis," in *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, Oct 2013, pp. 236–242.

[20] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults," in *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[21] Virtutech, "Simics Full System Simulator," Website, 2006, http://www.simics.net.

[22] 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, no. 2, pp. 1–7, Aug. 2011.

[23] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, J. Kalamatianos, O. Kayiran, M. LeBeane, M. Poremba, B. Potter, S. Puthoor, M. D. Sinclair, M. Wyse, J. Yin, X. Zhang, A. Jain, and T. G. Rogers, "Lost in Abstraction: Pitfalls of Analyzing GPUs at the Intermediate Language Level," in *International Symposium on High-Performance Computer Architecture*, 2017.

[24] A. Butko, F. Bruguier, A. Gamati, G. Sassatelli, D. Novo, L. Torres, and M. Robert, "Full-System Simulation of big.LITTLE Multicore Architecture for Performance and Energy Exploration," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, Sept 2016, pp. 201–208.

[25] R. de Jong and A. Sandberg, "NoMali: Simulating a Realistic Graphics Driver Stack Using a Stub GPU," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 255–262.

[26] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[27] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *International Symposium on Computer Architecture*, 1995.

[28] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "Accept: A Programmer-guided Compiler Framework for Practical Approximate Computing," in *Technical Report UW-CSE-15-01-01, University of Washington*, 2015.

[29] A. Mahmoud, R. Venkatagiri, K. Ahmed, D. Marinov, S. Misailovic, and S. Adve, "Leveraging Software Testing to Explore Input Dependence for Approximate Computing," in *Workshop on Approximate Computing Across the Stack (WAX)*, 2017.

[30] R. M. Stallman and G. DeveloperCommunity, *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3*. Paramount, CA: CreateSpace, 2009.

[31] M. G. Sobell, *A Practical Guide to Solaris*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[32] A. Chatzidimitriou and D. Gizopoulos, "Anatomy of Microarchitecture-level Reliability Assessment: Throughput and Accuracy," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2016, pp. 69–78.

[33] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2014, pp. 622–629.

[34] 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*, June 2014, pp. 375–382.

[35] N. Hasabnis and R. Sekar, "Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 311–324.

[36] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler-directed instruction duplication for soft error detection," in *Design, Automation and Test in Europe*, March 2005, pp. 1056–1057 Vol. 2.

[37] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-scalar Processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, Mar 2002.